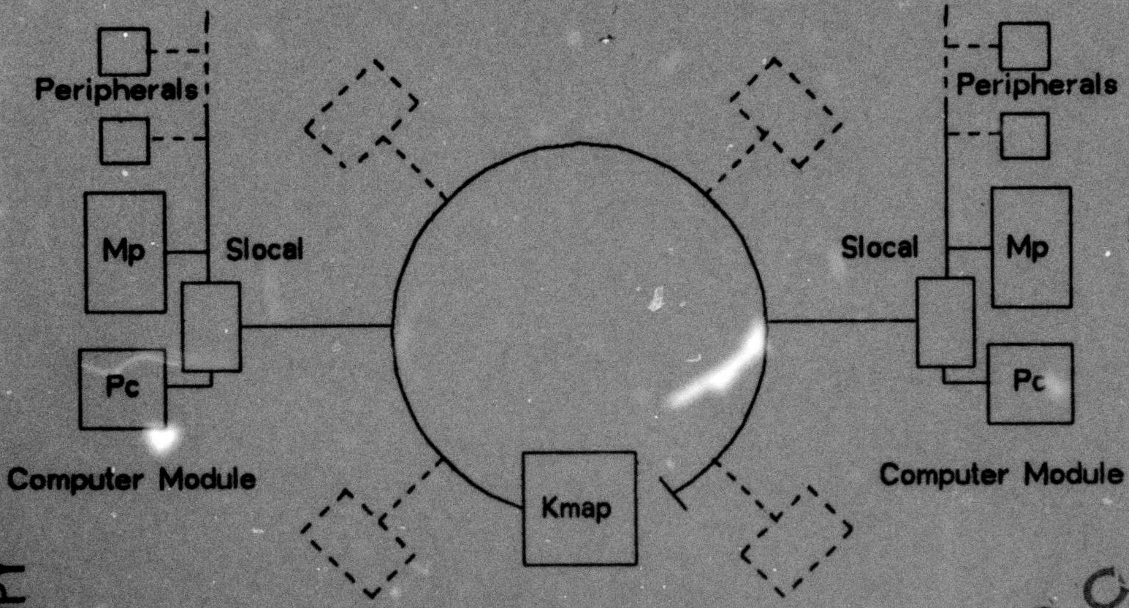
Carnegie-Mellon University Multi-Micropocessor Review

CMU Computer Science Department

Cm* Project



Cm* Review, June 1977

S.H. Fuller, A.K. Jones, and I. Durham (Editors)

Contributors:

Hardware Systems:	H. Bellis, S.H. Fuller, J. Ousterhout, P. Reddy, P. Rubinfeld, P. Sindhu, R. Swan.
Software Systems:	R.J. Chansler, I. Durham, P. Feiler, A.K. Jones, J.Ousterhout, D.Scelza, K.Schwans, S.Vegdahl
Algol 68:	P. Hibbard, A. Hisgen, T. Rodeheffer.
Benchmark Programs:	P, Feiler and L. Raskin.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract F44620-73-C-0074, which is monitored by the Air Force Office of Scientific Research, and in part by the National Science Foundation Grant GJ 32758X. The LSI-11's and related equipment were supplied by the Digital Equipment Corporation.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 78 - 0115 ✓ 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 CARNEGIE-MELLON UNIVERSITY MULTI-MICROPORCESSOR REVIEW.	5. TYPE OF REPORT & PERIOD COVERED 9 Interim rept., 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 S. H. Fuller , A. K. Jones ~~and~~ I. Durham	8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074, ✓ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department✓ Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101E A02466/7
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency/VM 1400 Wilson Blvd Arlington, VA 22209	12. REPORT DATE 11 June 1977 13. NUMBER OF PAGES 88 12 91P.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research/NM Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

18 AFOSR

19 TR-78-Ø115

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Cm* project has been in progress almost exactly two years. The present review is intended to critically examine our progress to date and evaluate our plans for the future of Cm*. ~~We will not review~~ the architecture, hardware implementation, or operating system designs in this report. The Cm* hardware and software designs were described in a session at NCC'77 [1,2,3]. This report contains a description of the measurement and evaluation studi~~es conducted~~, in the spring of 1977. Some of the data included here is preliminary.. However, given the

next page

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081 JOB

20. Abstract

it was desirable

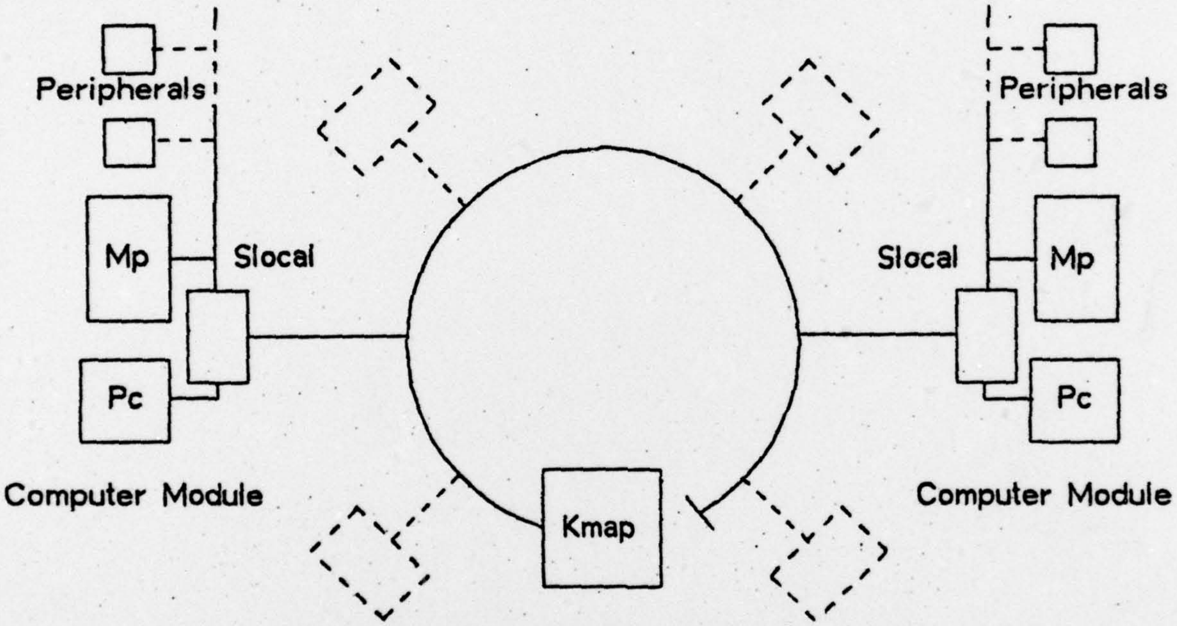
dearth of quantitative knowledge about multiple processor systems, ~~we wanted~~ to evaluate Cm* as early in its development cycle as possible to detect and correct flaws in our design and understanding of this experimental multiprocessor. We have made an effort to describe the present state of Cm* and ~~tabulate~~ all of our current measurements so a diligent reader can study and evaluate for himself our recent results.

tabulated

is described

CMU Computer Science Department

Cm* Project



Cm* Review, June 1977

S.H. Fuller, A.K. Jones, and I. Durham (Editors)

Contributors:

Hardware Systems:	H. Bellis, S.H. Fuller, J. Ousterhout, P. Reddy, P. Rubinfeld, P. Sindhu, R. Swan.
Software Systems:	R.J. Chansler, I. Durham, P. Feiler, A.K. Jones, J.Ousterhout, D.Scelza, K.Schwans, S.Vegdahl
Algol 68:	P. Hibbard, A. Hisgen, T. Rodeheffer.
Benchmark Programs:	P, Feiler and L. Raskin.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract F44620-73-C-0074, which is monitored by the Air Force Office of Scientific Research, and in part by the National Science Foundation Grant GJ 32758X. The LSI-11's and related equipment were supplied by the Digital Equipment Corporation.

Introduction to the Cm* Review

S.H. Fuller and A.K. Jones

The Cm* project has been in progress almost exactly two years. The present review is intended to critically examine our progress to date and evaluate our plans for the future of Cm*. We will not review the architecture, hardware implementation, or operating system designs in this report. The Cm* hardware and software designs were described in a session at NCC'77 [1,2,3]. This report contains a description of the measurement and evaluation studies conducted in the spring of 1977. Some of the data included here is preliminary. However, given the dearth of quantitative knowledge about multiple processor systems, we wanted to evaluate Cm* as early in its development cycle as possible to detect and correct flaws in our design and understanding of this experimental multiprocessor. We have made an effort to describe the present state of Cm* and tabulate all of our current measurements so a diligent reader can study and evaluate for himself our recent results.

The ten processor configuration of Cm* shown in Figure 1 is operational, and was the system from which the measurements described here were collected. In addition, a kernel operating system is also running on Cm* to provide the essential services for the larger application programs described later. No hardware-software system can be evaluated in isolation, and the following application programs have been used in our review of the prototype Cm* system:

1. An asynchronous algorithm for the solution of partial differential equations
2. Quicksort
3. Set partitioning via integer programming (e.g. the airline crew scheduling problem)
4. HARPY speech recognition system
5. Algol 68 runtime system (with facilities for some automatic detection and execution of concurrent operations)

Figure 2 shows the average speedups measured on Cm* as a function of the number of processors allocated to the task. These results are very encouraging. Some curves do not approach linear speedup. This does not exhibit any inherent limitation of Cm*; rather it illustrates inefficiencies in the algorithms being used. Sections 4, 5, and 6 of this report go into much more detail on the measurement and evaluation of those benchmark programs. Analysis of these programs indicates that the first order characteristics of their behavior can be understood from measurements of their memory access patterns and the following average times between successive memory references within Cm* (rounded to the nearest microsecond):

Time between local memory references	3 microseconds
Time between inter-Cm (same cluster) references	9 microseconds
Time between intercluster references	26 microseconds

For applications to make cost-effective use of Cm*, they must make most memory references to local memory. For the programs we have measured, the hit ratios are:

PDE's	97%
Quicksort	90%
Integer programming	98%
HARPY	87%
Algol 68	82%

Several operating system primitives that we expect to be used heavily have been implemented in Kmap microcode. The execution times for some of these are given below. (The approximate number of typical LSI-11 instructions which could be executed in the same amount of time are indicated in parentheses.)

Make segment addressable:	23 to 29 microseconds (4 instructions)
Synchronization primitive:	40 to 80 microseconds (6-12 instructions)
Transfer, Copy or Delete capability:	40 to 120 microseconds (6-18 instructions)

The operating system report (Section 3) also indicates those software implemented operations that we expect will be executed frequently. The interprocess communication operations, Send and Receive, currently perform below expectations because they do not yet have the microcode support originally planned. Consequently, other operating system operations such as process dispatching, which rely on Send and Receive are slower than desired. In the near future we expect to reexamine the software/microcode tradeoffs in these operations and find ways to improve their operation.

Our past experience with constructing and debugging large hardware and software systems caused us to pay particular attention to providing effective development aids and monitoring the reliability of the system. The availability of a Cm* simulator running on C.mmp permitted us to debug the operating system software before the hardware and firmware were usable by the software group. The Host computer along with the Hooks processors, provided support essential to the rapid development of Cm*. Section 8 documents the results of an automatic diagnostic system that ran on Cm* during the last several months when it was not otherwise in use. The diagnostic system helped to and identify faulty and marginal components. Significantly, hardware availability has not been a bottleneck during the Spring evaluation of Cm*. Moreover, the hardware was regularly partitioned to allow multiple user groups simultaneous access to private clusters within Cm*.

At the inception of the Cm* project, there existed serious questions as to the effectiveness of the Map and Linc bus schemes and the ability to effectively program Cm*. Our evaluation of the hardware and software system structure of Cm* during these past few months indicates that our basic design and implementation is sound. The areas that need further evaluation, expansion, and possible reimplementation before constructing and programming a larger Cm* configuration are:

A. Reexamine the addressing architecture for simplifications.
B. Review new hardware components for use in more cost-effective Cm* implementations.
C. Consider extension of Map Bus to allow more Cm's per cluster.
D. Review the intercluster communication protocol.
E. Determine how best to employ K.map microcode to support software systems.
F. Add mechanisms to the operating system to support Multi-cluster configurations
G. Determine the minimum amount of code that must be duplicated in each Cm while maximizing the local memory hit ratio.

We are encouraged by the results from our evaluation of the initial Cm* system. The difficulties and shortcomings that have been uncovered are within our original expectations. Studies are now in progress within the Cm* group on how to improve the existing Cm* design, identify the research problems associated with much larger Cm* systems, how to apply Cm* to one or several major applications, and to determine the effect of various component failures and to suitably adapt the system to permit continued processing.

Acknowledgements

While not involved in the work reported here, A. Bechtolsheim, K. W. Lai, and D. Siewiorek have made important, earlier contributions to the hardware design and implementation of Cm*. P. Oleinick provided many helpful suggestions in re-designing the HARPY speech system for Cm*. This report would not have been possible without the tireless editing of I. Durham.

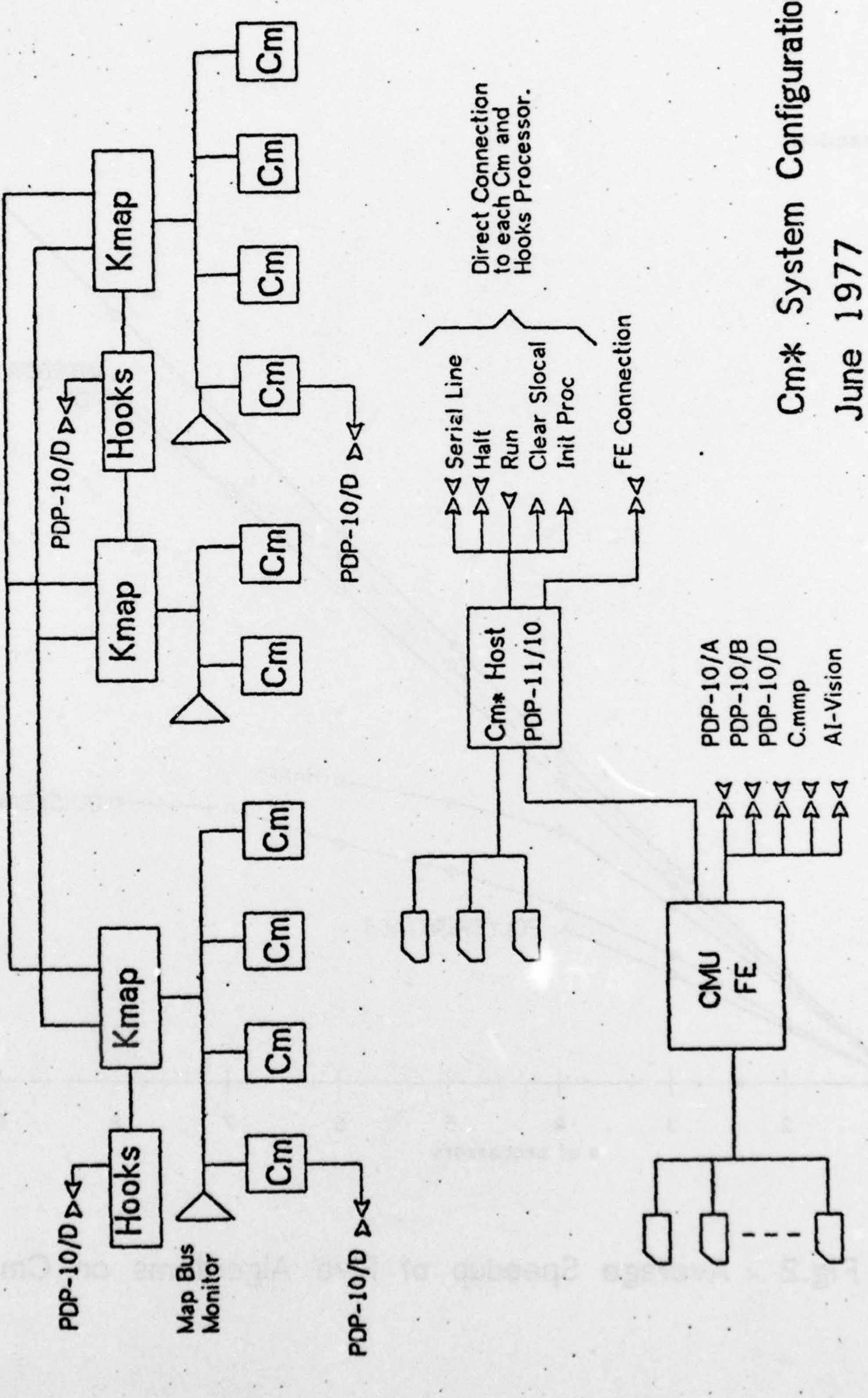
In addition to the ARPA, NSF, and DEC support, several students received support from other sources: H. Bellis, ONR, J. Ousterhout, NSF Fellowship, S. Vegdahl, Hertz Foundation, and L. Raskin, Israeli Fellowship.

References

[1] R.J. Swan, S.H. Fuller, D.P. Siewiorek, "Cm*: a Modular Multi-Microprocessor", NCC Proceedings, Dallas, Texas, 1977

[2] R.J. Swan, A. Bechtolsheim, K. Lai, J.K. Ousterhout, "The Implementation of the Cm* Multi-Microprocessor", NCC Proceedings, Dallas, Texas, 1977

[3] A.K. Jones, R.J. Chansler Jr., I. Durham, P. Feiler, K. Schwans, "Software Management of Cm*, a Distributed Multiprocessor", NCC Proceedings, Dallas, Texas, 1977



Cm* System Configuration

June 1977

Figure 1

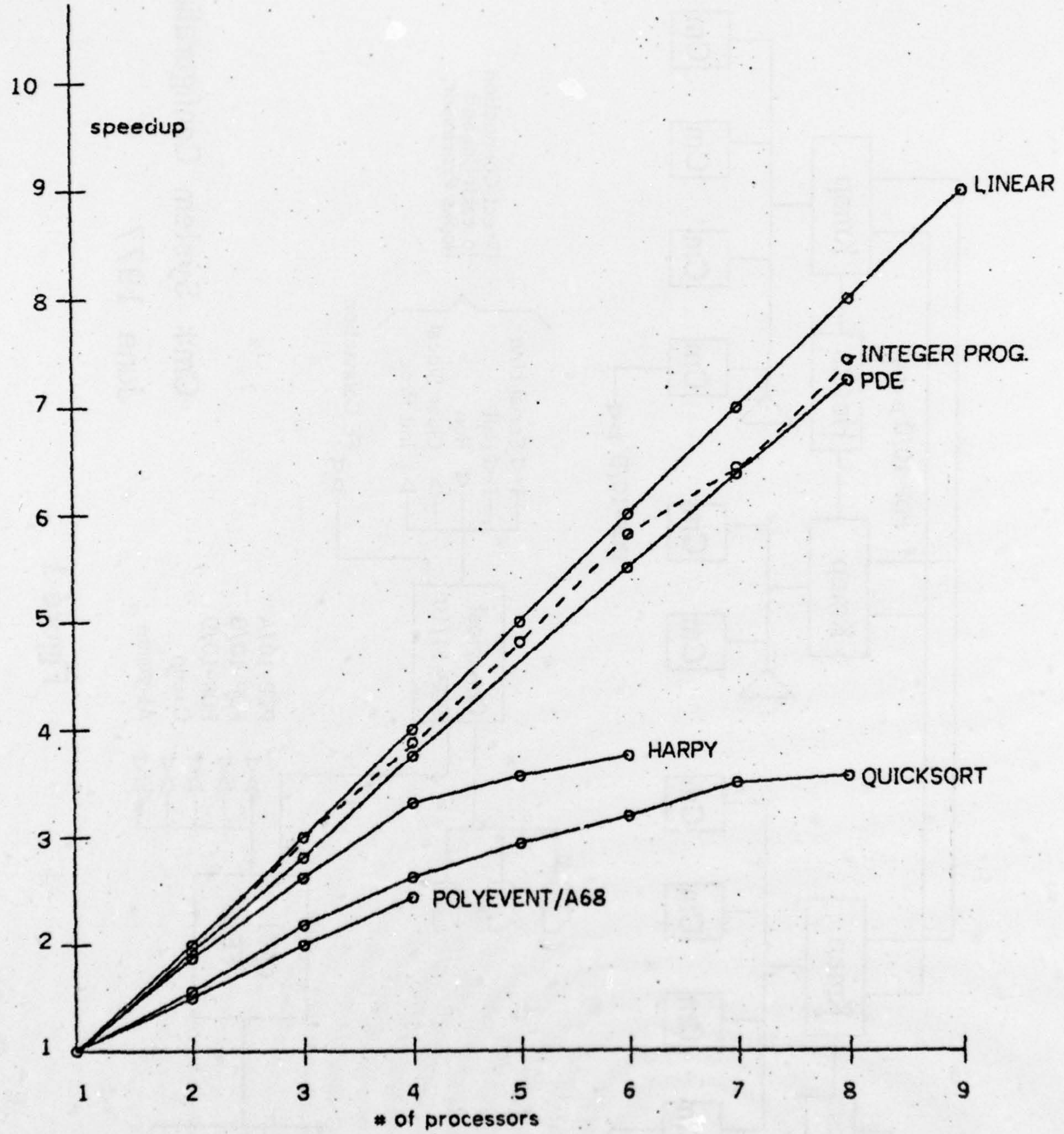


Fig.2 Average Speedup of Five Algorithms on Cm*

1. Hardware Status and Performance of Cm*

R. Swan

This report provides a brief, basic set of information about the hardware status and performance of the initial stage of the Cm* project. Figure 1, of the introduction, shows the overall system configuration. In addition to the present structure of Cm*, it shows the hardware provided for support and development. The complete set of hardware components available is listed in Table 1. A time-line of significant milestones in the Cm* project is shown in Figure 1. It indicates that approximately two years were required to reach the present fully operational status of a three cluster, ten processor system.

1.1. Basic Timings

The simplest unit for representing the performance of Cm* is total memory references per second. For PDP-11s, executing compiled BLISS-11 programs (Harpy, PDE and Quick Sort benchmarks), there are approximately 1.7 memory references per instruction. For a simple, synthetic program, the measured period between references to local memory was 2.9 microseconds. This corresponds to a memory reference rate of 348 KHz or about 0.20 MIPS (Millions of Instructions Per Second). Thus a ten processor Cm* system, where all memory references are local, is capable of providing up to a total of 2 MIPS. The following section provides some basic timing information [1] which shows how performance degrades when non-local (i.e. mapped) references are made under various conditions.

When a processor generates an address it may refer to local memory (the primary memory associated with that computer module) or it may be passed, via the Kmap, to the local memory of another processor within the cluster or passed over an intercluster bus to another cluster. Local memory references occur at the full rate of a stand alone LSI-11. Mapped references incur a delay through the switching

[1] Unless specifically stated, all timings apply to the simple version of the microcode. There are presently two versions of the Kmap microcode: a simple version which provides the basic address mapping necessary to access the memory of any processor in the system. The second version implements a segmented addressing scheme with capability based protection. See section 2 below. For a simple mapped reference with the full capability based microcode add 0.45 us to the times given.

structure. A detailed timing diagram for a single intracluster, mapped reference is shown in Figure 2. Figure 3 shows the various stages of an intercluster reference. From these two figures we see the following basic timings for memory references (with no contention from other processors):

> Time between references to local memory is 2.9 - 4.0 usec. (This is strictly a function of the LSI-11 processor, and depends on the instruction mix)
>
> Time between references to other memory in the same cluster is 9.3 usecs
>
> Time between references to memory in another cluster is 26 usecs.

Because of this wide range in accessing time for primary memory, the performance of Cm* depends strongly on the relative frequency of accesses to the different levels of memory.

1.2. Contention and Saturation Levels

The overall performance of a a Cm* cluster depends primarily on the percentage of references to local memory made by each processor. For the applications presently running on Cm* the 'hit ratio' to local memory is in the range 85% - 95% (see later sections of this report for details). Cluster performance is also effected by contention for the Map Bus and Kmap address mapping mechanisms and contention for shared memory. The graph in Figure 4 shows the approximate effective power of each processor for various hit ratios under a variety of conditions. The X axis is the hit ratio to local memory. An 80% hit ratio indicates that out of ten references, eight are to the local memory of the processor and two are two the local memory of another processor in the cluster. The Y axis indicates the performance of a processor, under the conditions specified, relative to a stand alone LSI-11. The plot for a single Cm indicates the degradation in performance strictly as a consequence of the delay in accessing non-local memory. For example, for a 90% hit ratio a processor has the effective power of 0.85 of a stand alone LSI-11. The plot for 8 Cms (no memory contention) shows the influence of contention for the Map Bus and Kmap. This contention has only a small effect on performance with 8 Cms. With a hit ratio of 90% the additional degradation is about 4%. For a hit ratio of 50% the additional degradation is about 7%. The third plot represents the case of 8 processors sharing a non-local data structure, which resides in a single Cm. For low hit ratios (frequent references to shared data) memory contention severely degrades performance. With 8 Cms the shared memory is close to saturation for hit ratios below 85%. However, at a 90% hit ratio, memory contention has reduced performance by less than an additional 2%.

Table 2 lists overall cluster performance both as a function of the number of Cms and the hit ratio to local memory. As with Figure 4, the effect of sharing non-local memory is also shown. For example, the table shows that 6 Cms, with an 85% hit ratio, are equivalent to 4.6 LSI-11s when the non-local data is shared and 4.7 LSI-11s if there is no memory contention. The overall conclusion from these measurements is that an 8 Cm cluster, operating in the anticipated region of an 85% - 95% hit ratio, executes instructions at the same rate as 5.7 to 7.2 individual LSI-11s.

These measurements were obtained using small synthetic programs. Later sections of this report present similar data based on measurements of application programs. An important aspect of the hardware performance is the maximum capacity of various components. The saturation levels may be summarized as follows:

Processor's normal reference rate to local memory	348 KHz
Maximum total memory reference rate over the Map Bus	500 KHz
Maximum mapped reference rate to same memory	250 KHz

1.3. Measurement Tools

A small set of simple performance evaluation tools were developed to provide the measurements presented in this and other reports. The tools include:

1. Map Bus Monitor. This is a simple display which gives access to control information and data passed over the Map Bus. Any specific event can be detected by masking against a set of switches corresponding to each Map Bus signal. Information gathered is latched in a set of lights for inspection by a programmer or engineer and a match signal is fed to an external frequency meter for counting and averaging.

2. Microcode Analysis. A standard logic analyzer is connected to the internal microinstruction address lines of the Pmap. The match signal from the analyzer is connected to an external frequency meter. This technique allows counts of the frequency of invocation of various Kmap operations, for example: simple mappings, load segment, copy capability, increment semaphore, etc.

3. Measurements within a single Computer Module. Frequency meters were connected to important signals generated by the processor and other components of a Cm. For example, local memory hit ratios are determined by measuring the overall memory reference rate of a processor and comparing it with the mapped memory reference rate, measured using the Map Bus Monitor.

Table 1

Hardware Components of Cm*

Host System

PDP-11/10
28K Memory.
Hardware Bootstrap.
4 DECtape drives.
DJ11/16-Serial Line Unit (SLU) Multiplexer connected to LSI-11s and local terminal.
4 SLUs: 2 connected to the Front End & 2 for local terminals.
2 DR11-Cs: Halt, Reset etc. for Computer Modules.

Computer Modules

Modified LSI-11 Processor.
Slocal. Provides local relocation, Map bus interface etc.
Memory Refresh and LSI-Bus Terminator.
Serial Line Unit (SLU). For communication with the Host.
Power Board. Power connections, initialization etc.
Memory. 4 - 28K words. Capability for 124K words.
Parity Board (under development.)

Computer Modules Available

2 with 24K words of memory.
8 with 28K words of memory.

Kmaps (Central Mapping Controllers)

3 Available with:

Kbus. Map Bus Controller and Pmap Dispatcher. 100 ns cyole time.
Pmap. 150 ns cycle time, microprogrammed processor
Data RAM. 5K x 16 bits fast mapping table storage
Writable Control Stores: 2K x 80 bits
Linc. Intercluster Bus interface
Hooks Processor Interface

Hooks Processors

2 Available, used for loading and controlling Kmaps:
LSI-11 Processor with 20K words of memory.
3 SLUs, direct communication to CMU-10D and Host
Hooks Pc #1 controls Kmap #1
Hooks Pc #2 controls Kmaps #2 and #3

No. of	Hit Ratio to Local Memory							
Processors	50%	60%	70%	80%	85%	90%	95%	100%
1	0.54	0.59	0.66	0.75	0.81	0.85	0.88	1.0
2	(1.0 - 1.1)	(1.2 - 1.2)	(1.3 - 1.3)	(1.4 - 1.5)	(1.5 - 1.6)	(1.6 - 1.7)	(1.8 - 1.9)	2.0
3	(1.4 - 1.6)	(1.7 - 1.7)	(1.9 - 1.9)	(2.2 - 2.3)	(2.3 - 2.4)	(2.5 - 2.5)	(2.7 - 2.7)	3.0
4	(1.8 - 2.1)	(2.1 - 2.3)	(2.5 - 2.6)	(2.9 - 3.0)	(3.1 - 3.2)	(3.4 - 3.4)	(3.7 - 3.7)	4.0
5	(1.9 - 2.6)	(2.3 - 2.8)	(2.8 - 3.2)	(3.5 - 3.7)	(3.9 - 3.9)	(4.2 - 4.2)	(4.6 - 4.6)	5.0
6	(1.9 - 3.1)	(2.4 - 3.3)	(3.1 - 3.8)	(4.0 - 4.4)	(4.6 - 4.7)	(5.0 - 5.1)	(5.4 - 5.4)	6.0
7	(1.9 - 3.4)	(2.4 - 3.8)	(3.2 - 4.4)	(4.4 - 5.1)	(5.2 - 5.5)	(5.8 - 5.9)	(6.3 - 6.4)	7.0
8	(2.1 - 3.7)	(2.6 - 4.2)	(3.4 - 5.0)	(4.8 - 5.7)	(5.7 - 6.2)	(6.5 - 6.6)	(7.2 - 7.2)	8.0

Description: Relative Processing Speed , 1.0 = One stand alone LSI-11
(all non local references to same physical memory - all non local references to different memories)
All processors are executing ADD R0, R0 (4 us)
In the region below the staircase line the shared memory is saturated.

Table 2. Relative Processing Power, with and without shared non local memory.

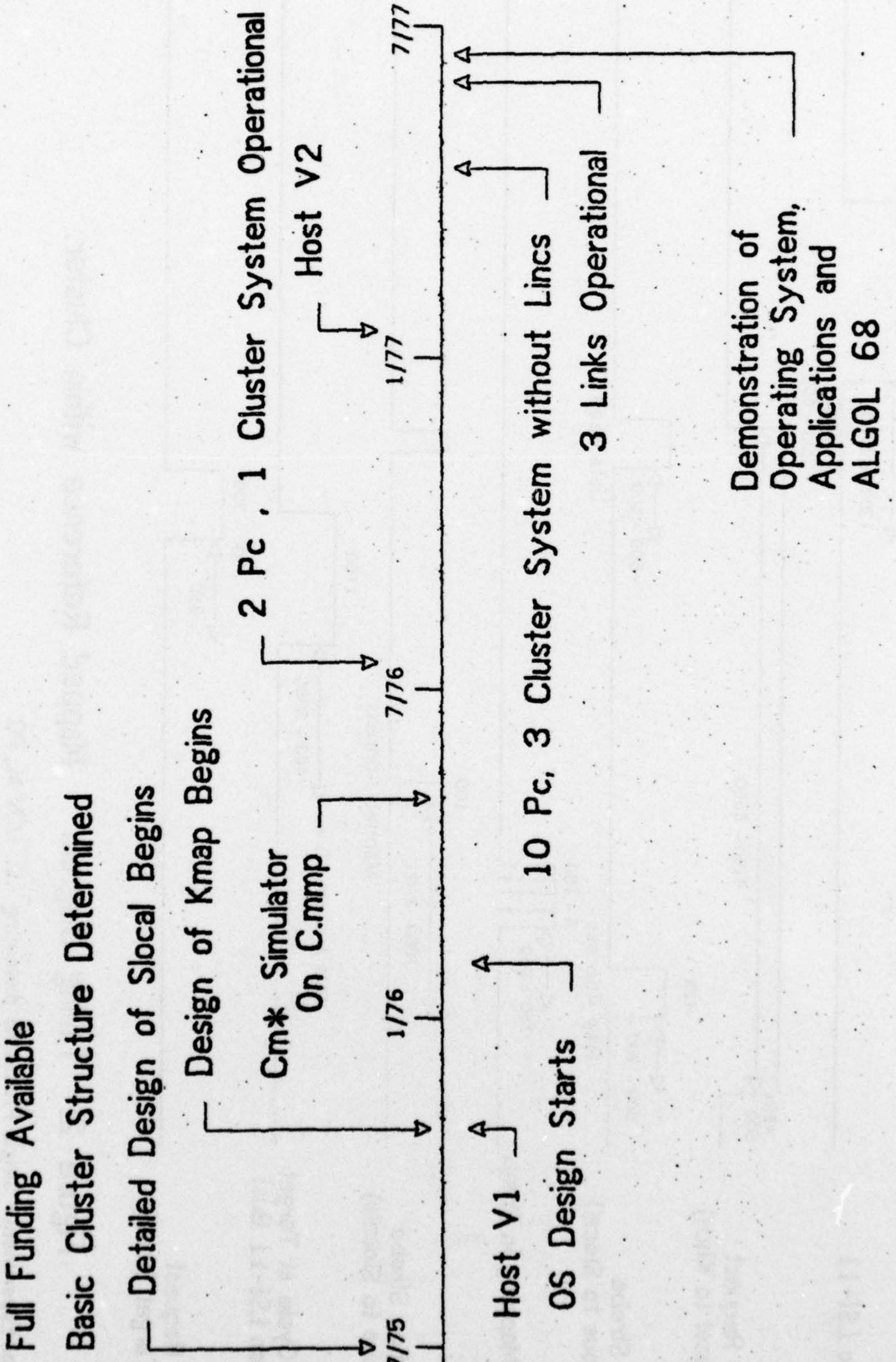

Full Funding Available
Basic Cluster Structure Determined
Detailed Design of Slocal Begins
Design of Kmap Begins
Cm* Simulator On C.mmp
2 Pc , 1 Cluster System Operational
Host V2
7/75
1/76
7/76
1/77
7/77
Host V1
OS Design Starts
10 Pc, 3 Cluster System without Lincs
3 Links Operational
Demonstration of Operating System, Applications and ALGOL 68

Figure 1 The Development of Cm*

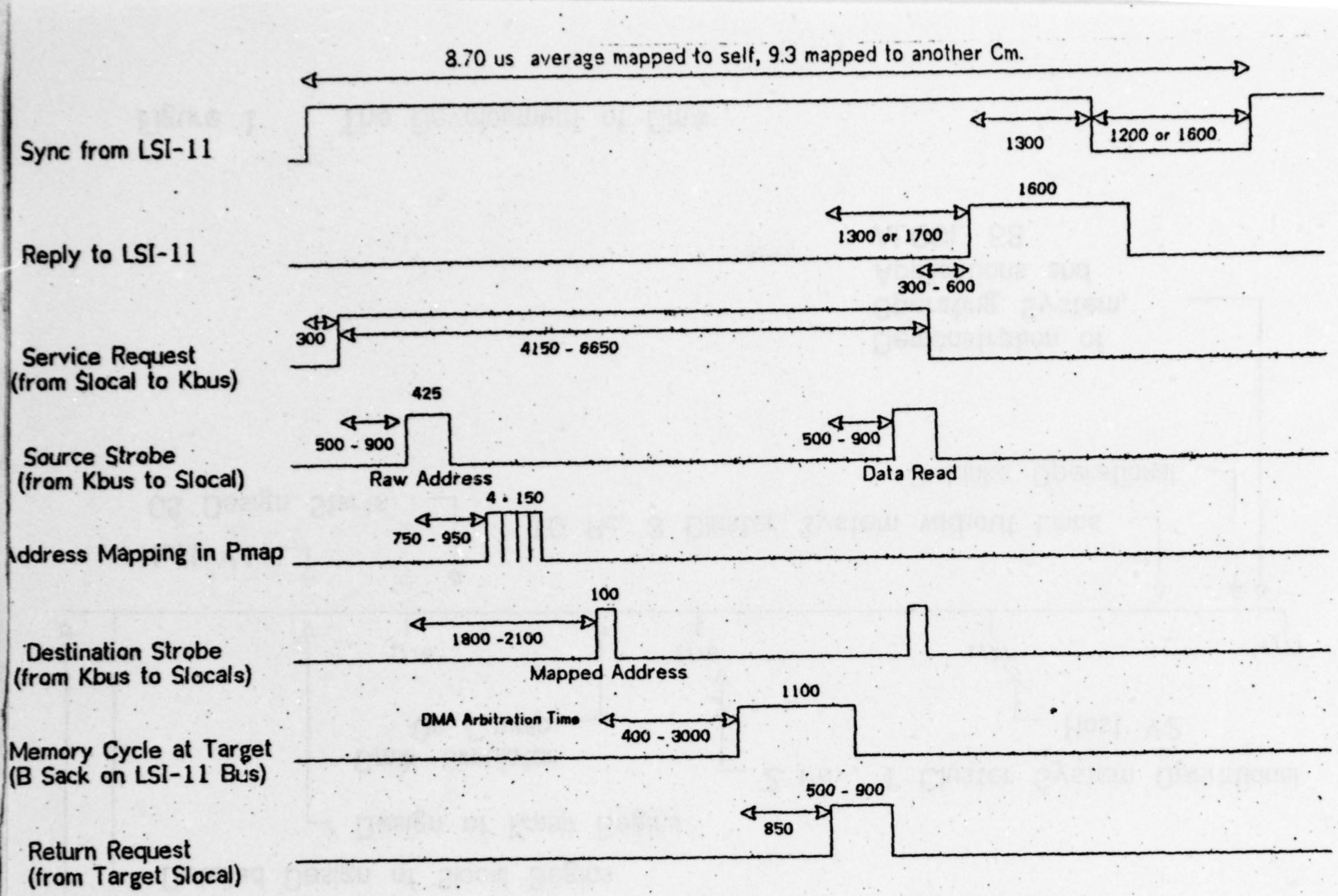
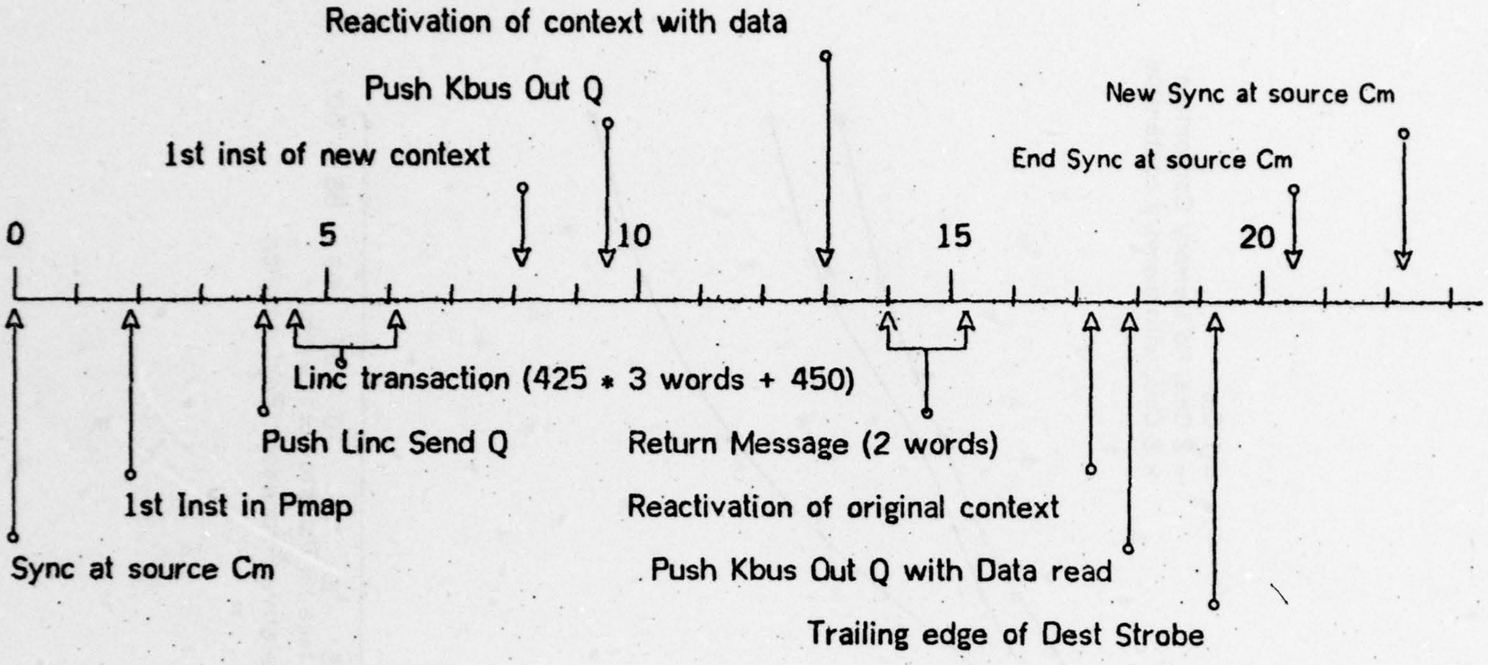


Figure 2: Timing Diagram Mapped Reference within Cluster.

Details of Experiment: Single Processor executing L: MOV #L, PC
Inter-reference time running all local is 2.9 us.
Shown Mapped back to the same Cm (8.7 us)
When mapped to a different Cm the DMA arbitration time is slower (9.3 us)
Cm1 in Cluster 1, May 30, 1977


Reactivation of context with data
Push Kbus Out Q
1st inst of new context
New Sync at source Cm
End Sync at source Cm
0
5
10
15
20
Linc transaction (425 * 3 words + 450)
Push Linc Send Q
1st Inst in Pmap
Sync at source Cm
Return Message (2 words)
Reactivation of original context
Push Kbus Out Q with Data read
Trailing edge of Dest Strobe

Inter-Cluster Reference: Reference mapped back to same Cm.

Average Period between references: 22.4 us. (2.9 us all local)

Kmap 1, May 26, 1977

Figure 3

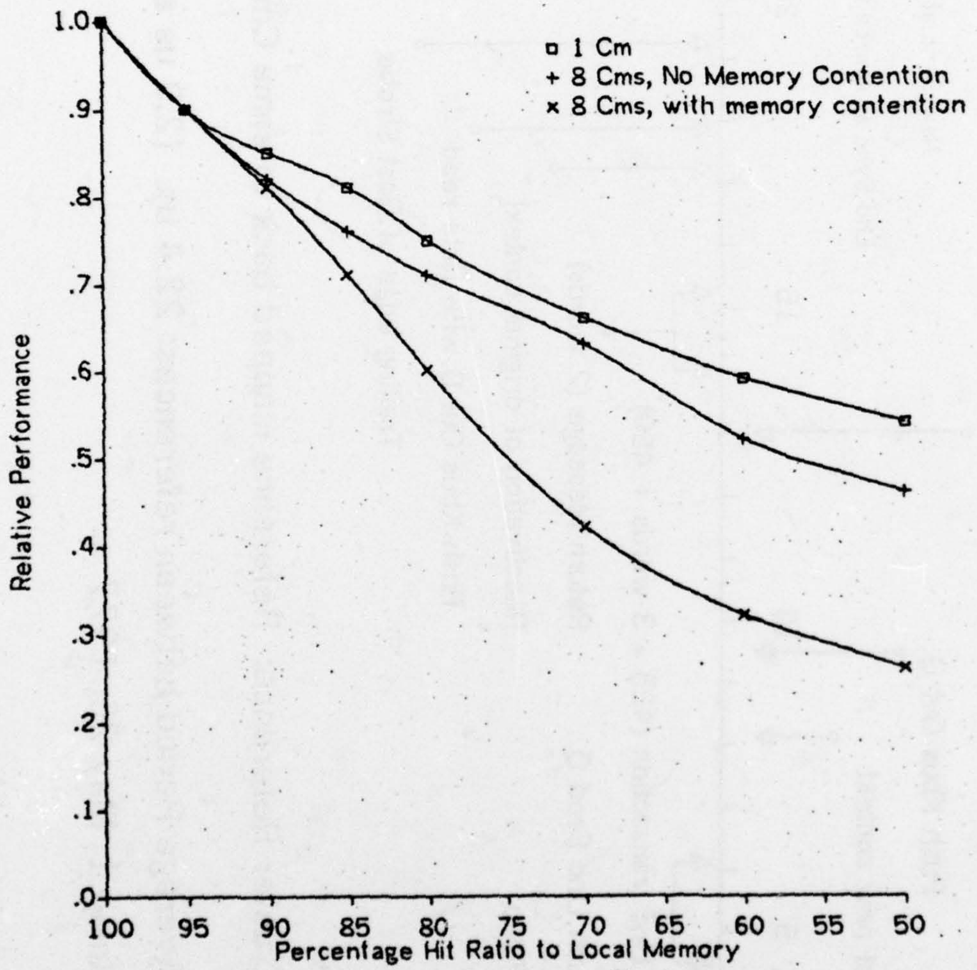


Fig. 4. Relative Processor Performance

2. Kmap Microprograms

J. Ousterhout

A substantial amount of microcode has been written for the writable control stores of the Kmaps. Development of microcode has proceeded along two independent paths; the results presented in this review reflect both of the versions. The first version, referred to as the simple microcode, was produced in order to supply the bare minimum facilities needed to allow processors to share a single address space. No attempt was made to supply protection or elegance in address space manipulation. This version has been operational for nine months and has been used extensively by diagnostic programs and by Raskin's benchmarks and the Algol 68 system. The second version of microcode supplies the full Cm* virtual memory structure to a single-cluster system, complete with address space management and protection. This VM microcode has been in use for approximately two months.

2.1. The Simple Microcode

The principal feature supplied by the simple microcode is to allow each processor to associate any 2048-word physical page in the system (identified by its cluster number, Cm number within the cluster, and high-order six bits of its base address) with any page in its virtual address space. All references to the virtual page are then mapped to the physical page, with the low-order twelve bits of the processor address being concatenated to the page's base address. All pages contain 2048 words. When a processor references a page residing in its local memory it may select whether or not references to the page are to be made directly through the Slocal or mapped (for diagnostic and measurement purposes) through the Kmap.

References to page 17_8 are always passed to the Kmap and are not mapped in the same way as the other virtual pages. The Kmap responds to fifteen word addresses in this page, emulating a small memory which contains the relocation tables for the processor (these locations are called window registers). Page 17 references to locations other than the window registers are passed back to physical page 17 of the processor that made the reference so that it maintains access to all of its input/output devices. Although each processor has access to only its own window registers it may address any physical page in the system without restriction, thus the system is not a protected one.

Because it is not possible to perform non-local indivisible read-modify-write sequences the simple microcode implements a single lock that is global to the whole cluster. Reads to address #177776 cause the value of an internal location in the Kmap to be returned and then set the value to all ones. Writes to the location overwrite its value. More recently a major expansion of this microcode has been completed which provides a substantially cleaner and more general semaphore scheme; however the newer version was not available at the time measurements were made for this review. When multiple semaphores were required, the global lock was used to provide indivisibility while manipulating the multiple semaphores.

The simple microcode requires slightly less than 170_{10} words of control store in its simplest single-cluster form, of which about 50 words are used for initialization only. Including the expansions for multi-cluster service and more adequate synchronization the simple microcode grew to a size of 505 words. The breakdown of microinstruction usage is:

Function Performed	Instructions
Within-cluster references	11
Cross-cluster references	91
Window register references	11
Within-cluster synchronization	68
Cross-cluster synchronization	88
Error handling	82
Initialization	106
Page 17 decode and miscellaneous	48
Total	505

Microcode tends to have a very high branching factor due to the number of conditions being tested, so that typical microinstruction sequences are substantially shorter than the static number of instructions allocated to a particular function. Sequences range from five microinstructions (at 150 nanoseconds per microinstruction) for a simple within-cluster mapped reference to a total of about 60 instructions (for source and destination Kmaps combined) for an intercluster synchronization.

The simple microcode uses only a small portion of the Kmap's high-speed data store (which contains a total of 1024 records each with 5 16-bit words). 512 words are reserved for the window registers of each of the two address spaces per Cm (up to 14 Cms per cluster), and 256 more words are used to hold internal locks for synchronization operations.

2.2. Virtual Memory Microcode

The VM microcode implements the highly-protected and elegant addressing architecture described in Cm* publications [1-3 in introduction]. The version currently operational does this only for a single-cluster system and does not include message operations whose implementation was originaly intended for the Kmap. However it does enforce environment boundaries and perform much of the work involved in maintaining the integrity of the Cm* virtual address space.

Features of the VM microcode fall into three rough categories. First, execution environments may make direct (i.e. read and write) references to two different types of segments, where the references cause different actions for the different segment types. Secondly, they may make indirect references to a variety of segment types by writing the address of a parameter block into a special page 17 location monitored by the Kmap. Lastly they may invoke control operations which change the virtual-to-physical address translation of an environment.

Two segment types may be referenced directly. Data segments may be read and written at will, and stack segments may be read and written, but with the Kmap maintaining an invisible stack pointer and altering the actual address to be written or read in order to enforce a stack discipline (the segment appears as a single memory location which is the top element of a push-down stack; reads cause "pop's" and writes cause "pushes"). In addition to standard reads and writes the Kmap provides for indivisible increments and decrements of words in data segments.

Indirect references to segments are performed by passing to the Kmap (by writing its address into a location in page 17) the address of a parameter block containing the index of a capability and the description of an operation to be performed on the associated segment. All of the operations that may be invoked by direct references may also be invoked indirectly (the difference being that in the latter case the segment being referenced need not be loaded into a window register). Additionally several operations are defined on capability list segments and directory segments (the latter contain the descriptors for segments). These may only be performed indirectly.

The third service provided by the VM Microcode is to implement high-speed and protected operations to change the system addressability. The first such operation provided is to allow environments to change the binding between their virtual address space and physical memory by passing to the Kmap the index of a capability and the number of a virtual page with which the segment referred to by the capability is to be associated (this is referred to as a window register load). The second group of operations allows the high-speed loading and dumping of the internal Kmap state associated with a running of an environment, in order to speed up context swaps (state consists of the sixteen-bit values in each of the fifteen window registers of the environment and eight words of error status).

When an external reference is made by an environment the Kmap must access three items to map the reference: the window register, which contains the index of the capability associated with the page being referenced; the capability named in the window, which indicates the name of the segment being referenced and a set of rights thereon; and the descriptor for the physical segment. Windows are allocated statically, occupying 3 subwords of the first 512 records of the Kmap's data store. Capabilities and descriptors reside principally in main memory, but are cached in the Kmap in order to provide greater speed in performing external references. Records 512-1023 of the data store are used for cached capabilities and descriptors (intermixed) with either item requiring a full five-word record for its cache entry. In addition subwords 3 of the first 256 records contain cache list pointers.

To locate a cached entry 7 bits are extracted from the low-order portion of the unique name for a capability or descriptor and used to look up a list pointer in records 0-127 (for descriptors) or 128-255 (for capabilities). The pointer indicates the first element in a linked list of cache entries with the same low-order bits. The high-order bits of the name of the desired item are then compared with the high-order bits of the names of each of the items in the list for a match. When one cache element must be thrown out in order to create enough room to bring in another, a pseudo-LRU scheme is used. In this method a use bit is kept for each element and set whenever the element is used. The throw-out routine scans the cache, turning off use bits and throwing out the first element found that already had its use bit turned off from a previous scan.

The use of a dynamically-allocated cache required the writing of a substantial amount of microcode and incurred a sizable execution time overhead, but made possible a fairly substantial speedup in the time required for changing addressability. A cache "miss" when looking for a capability requires between ten and fifteen microseconds (two main-memory references) to read in the capability, assuming all of the information needed to locate the capability is already present in the cache. A descriptor miss involves nearly twenty microseconds (three references) in the best case. A worst case series of cache misses could require 22 main memory references (around 150 microseconds) before a window register could be bound to a physical address. (The breakdown of this is as follows: two descriptors, one for a directory and one for a capability list, in order to locate the primary capability for the window; then two more descriptors after reading in the primary capability to locate a secondary capability; then a directory descriptor and the actual segment descriptor)

The VM microcode requires 1515_{10} microinstructions in its current form. A rough breakdown of the static control store allocation by function is as follows:

Function Performed	Instructions
Descriptor cache manipulation	100
Capability cache manipulation	143
Window register manipulation	197
Direct data segment operations	64
Direct stack segment operations	28
Page 17 decode and control registers	139
Utilities for indirect operations	82
Indirect data segment operations	52
Directory operations	122
Capability operations	268
State loading and dumping	99
Error handling	123
Initialization	53
Miscellaneous utilities	45
Total	1515

The dynamic microinstruction sequences are somewhat longer in the VM microcode than for the simple microcode. The simplest mapping requires 8 microinstructions; a typical window-register load causes the execution of about 80-100 microinstructions; and a transfer capability operation involves 200-300 microinstructions. All of the above numbers assume a 100% cache hit ratio.

For the measurements reported later in the other sections of this report, the full cache was not used; its size was reduced to 32_{10} words.

3. Measurements of the Cm* Kernel/Machine

K. Schwans

3.1. Introduction

In this section we present the initial performance measurements of those Operating System functions we expect to be most frequently used. Before explicating the measurements we will characterize the scenario in which they were made.

The Operating System is distributed, i.e. there is no centralized control, there are no master - slave relationships, and processors execute autonomously. Only a few benchmark applications have been programmed to use the Operating System. Consequently, we have relatively little experience with the way users will employ the Operating System. However, we expect heavy use of certain functions. In particular, we do know that the Operating System relies heavily on the capability operations in terms of which a process' addressing environment is defined. In this initial evaluation we focus on the cost of individual operations involved in changing the addressing environment of a process, multiplexing a processor among processes, synchronization, and interprocess communication.

Cm* provides two media in which to implement an Operating System function: software and Kmap microcode. To invoke an operation implemented in software requires trapping to the kernel so that the operation can be executed in kernel space, then returning to the user. In contrast, microcoded operations are invoked through a special set of locations in each processor's virtual address space. Invocation and execution can be expected to be quite fast, because neither a processor trap nor fetching of instructions from main memory is required to perform the operation. Microcoded operations include synchronization primitives and operations for manipulating addressing state (for example, all capability operations).

Our original design specified that message communication operations also be implemented in microcode. Because microcode space was becoming scarce and because the operations could more rapidly be implemented in software, we decided to implement the message communication operations in software. We expect users to make fairly extensive use of this disciplined communication mechanism; in addition, the Operating System itself relies on message operations to perform multiplexing. Since we are concerned that multiplexing be a relatively efficient function, multiplexing measurements are reported below.

We maintain local copies of some amount of kernel code. A processor executes code locally if possible, otherwise it must execute remotely. Because execution of remote code is significantly more expensive than execution of local code, issues arise, such as:

i. What amount of kernel code should be local to each processor for good performance?

ii. Where is the tradeoff between remote code execution and moving the locus of control to a processor with local code?

As of now, 4,096 words of kernel code are local to each processor and all other kernel code is executed remotely. Included in the local kernel code are the message operations, the multiplexing code, the kernel entry/exit code, and interrupt/trap handling routines.

3.2. Measurement Techniques

Two kinds of measurements were performed:

3.2.1 Memory reference counts

Counting memory references of frequently used Operating System code, we have made a number of reasonable assumptions not explicated in every case. They are control flow assumptions such as:

i. Mailboxes are not empty. (If a Conditional Receive is performed on a mailbox, it does not fail.)

ii. Capability parameters to a system call are located in primary or secondary capability lists.

The <u>rough time estimates</u> are derived by code counts, assuming:

i. Code and stack local to the executing processor

ii. Data residing in another module

The processor reference time for local references is assumed to be 2.9 microseconds. Mapped references are assumed to take 9.3 microseconds.

3.2.2 Measurements on the actual hardware

The measured times were taken on the hardware under a slightly optimized version of the OS. The OS had all kernel entry/exit code local to each processor, and no kernel code contained debugger or Kmap tracing calls. Error checking of each Kmap call was still performed. This constitutes an overhead of one test/branch per Kmap call.

The following three methods were employed for taking the measurements under artificial Kmap load conditions:

i. A tight loop executes on one processor, while all others are halted. Thus the Kmap has only to service requests from one module. The measurements are taken with the logic analyzer and the Kmap bus monitor concurrently.

ii. A larger loop, generally about 20 code references, 10 stack references, and two mapped data references executes on one module, while the others require continual Kmap service (busy waiting on I/O).

iii. A loop (as in ii) executes on one processor, while the other processors require Kmap service infrequently (approx. every 20,000 instructions).

Timings obtained with methods ii,iii include a potential error of up to 10% due to the necessary estimation of the loop execution times. Additional inaccuracy (approx. 1%) was introduced by the use of a low resolution timing device. This error was minimized by repeating the operations 30,000 times and averaging the results.

3.3. Software Measurements

3.3.1 Saving and loading the processor state

Load: Make the designated user the *current* user, set the current user's state to *active*, and load the current user.

Rough Time Estimate:

```
  124 Microseconds processor execution time
+ 150 Microseconds Kmap execution time
  274 Microseconds total
```

Unload: Reverse the above process.

Rough Time Estimate:

```
  112 Microseconds processor execution time
+ 135 Microseconds Kmap execution time
  247 Microseconds total
```

3.3.2 Cost of dispatching a process

Here we assumed that the processor was not idle. This ensures that the cost of unloading an environment is included. The dispatching algorithm may briefly be described as follows:

```
Test whether processor is idle or not.
IF Idle
  THEN  Unload current environment;
        SEND environment to its Run Queue.
FI
WHILE True DO
  Wait a while
  ConditionalReceive a runnable environment.
  Load the new environment
OD
```

Rough Time Estimate:

498	Microseconds processor execution time plus the cost of one Conditional Send and one Conditional Receive on the processor runqueues.
+ 350	Microseconds Kmap processing time
848	Microseconds total plus the two Message Operations

Measured Time: 4.73 milliseconds (including Message Operations)

As can be deduced from the measured times and the message operation costs shown below, the latter dominate the cost of dispatching a process.

3.3.3 Message Operation costs

3.3.3.1 Measured Time

Conditional Send:	2.03 Milliseconds (local stack) 2.72 Milliseconds (mapped stack)
Conditional Receive:	2.03 Milliseconds (local stack) 2.72 Milliseconds (mapped stack)

Users executing the software message operations have to perform a kernel call. Thus additional costs are incurred.

3.3.3.2 Total costs for the user

Conditional Send:	5.24 milliseconds
Conditional Receive:	5.24 milliseconds

3.3.3.3 Total cost in case of firmware implementation

Our best estimate of execution time for a Conditional Send or Receive in case of Kmap implementation is: 200 microseconds.

3.3.4 Dispatch cost with microcoded message operations

Using the 200 microsecond estimate from above, the cost of a dispatch then becomes:

Memory operations	498 Microseconds	40%
Capability operations	350 Microseconds	30%
Message operations	400 Microseconds	30%
Total:	1248 Microseconds	

3.3.5 Costs of kernel entry/exit

The following steps are involved in kernel entry and exit:

Save and restore the registers
Save and restore the user PC,PS
Determine the EMT number
Select the action
Move the parameters
Return from the EMT back to user space

Rough Time Estimate: (n = # of parameters)
2.02 + n*0.2 millisec plus Kmap operation time. Note that the Kmap operations performed in the entry/exit routines are not included.

Measured Time: 3.14 milliseconds (call with two parameters)

3.3.6 Further Software Measurements (intended)

1. Process creation system calls

   Explicitly: try to compare building a new environment as opposed to retrieving one from a pool.

2. Memory operations:

   i. Performance of bitstring allocation strategies

   ii. Createsegment costs, using Kmap directory operations and distributed memory management data structures

3. Overlay costs

   i. Overhead for overlaying segments in the kernel

   ii. Patterns of overlay, optimal strategies

3.3.7 Interaction and Contention (intended)

1. Contention in the message system - two modules send/receive messages in various patterns with various frequencies

2. Locking overhead and waiting times on shared system data structures

3. Compare the cost of using the message system to using shared segments with explicit locking.

4. Contention in the runqueues for dispatching

5. Intensive process swapping experiments

6. Optimal communication of data between processes:

  i. Referencing shared segments in remote modules as opposed to

  ii. Copying segments to enable local referencing

7. Compare cost of:

  i. Remote code execution,

  ii. Moving a process to the processor that can execute the code locally,

  iii. Copying code segments into local memory

3.4. Firmware Measurements

3.4.1 Firmware (i.e. operations microcoded in the Kmap)

3.4.1.1 The parameters

The parameters for the measurements of the capability and locking operations are:

  i. Access to a primary or secondary capability list (directly/indirectly)

  ii. Capability accessed can be expected to be cached or not (cached/not cached)

  iii. Kmap contention caused by requests from other processors (contention/no contention)

The ranges given result from variation in the parameters. The typical times are underlined.

3.4.1.2 The functions

i. LoadSegment (make a segment directly addressable) - 23 to 39 microseconds

ii. ReadWord, WriteWord (read and write to a segment which is not directly addressable) - 40 to 80 microseconds

iii. ReadCapability (read the capability contents) - 40 to 114 microseconds

iv. TransferCapability (move a capability from one slot to another) - 70 to 145 microseconds

v. CopyCapability - (estimated cost) 127 microseconds

vi. DeleteCapability - (estimated cost) 89 microseconds

vii. IncrWord, DecrWord (synchronization instructions) - 40 to 80 microseconds

3.5. Conclusions

An Operating System can only be evaluated on the basis of the services it provides to its users. We are not yet at that stage of evaluation. However, we believe that we have been successful in designing primitives that are useful both for application programs and the Operating System. The primitives are simple enough to be microcoded and are cost effective in that we derive substantial improvement in execution speed using the microcode implementation. We have gained some insight into the use made of these operations by the Operating System, but more evaluation is required to make optimal use of the centralized microcode resource.

At the design stage of the Cm* project, we speculated that operations that were designed to be implemented in microcode could be implemented in software with no functional change. The proved to be the case with the message operations.

In this evaluation emphasis has been on the microcoded Operating System operations that allow programmers to make maximal use of the Cm* resources. One of the basic problems of the PDP-11 machine is the small 16 bit address space. We have overcome this problem by providing a capability addressing space together with efficient capability space manipulation operations. In addition, Cm* provides efficient operations to make segments directly addressable or to read/write arbitrary words out of/into segments in the virtual address space. The execution times of these operations range from 23 microseconds for performing a 'LoadSegment' operation to 130 microseconds for a 'CopyCapability' operation. This compares very favorably to the average LSI-11 instruction execution time of 6 microseconds or the floating point multiply cost of 94 microseconds.

Processes can communicate through shared segments or by messages. The explicit synchronization instructions (IncrWord/DecrWord) provided for locking shared data segments, also implemented in Kmap microcode, cost only 40 microseconds. Segments do not have to be directly addressable to perform locking operations on them, thus no overlay costs are incurred.

The message operations (costing 2 milliseconds each) are slower than we would like. If microcoded, the operations are estimated to cost 200 microseconds each, a time that would also compare favorably with LSI-11 instructions. We believe that reducing the cost of the message operations by an order of magnitude would substantially increase their usage. Certainly, in the Operating System itself the message operations dominate the cost of multiplexing environments. Almost all the time in executing a 'dispatch' is spent there. Either the multiplexing function will be redesigned so that it does not rely on message operations, or the message operations will have to be microcoded to increase the speed of a 'dispatch' operation. Users executing the current message operations have to perform a costly kernel entry/exit. Thus the costs are twice as high for the user as for the system. In a microcoded implementation this anomaly will not persist.

Further measurements are in progress. The process creation system functions, the memory manager operations, current kernel overlay costs, and interaction and contention effects will be studied. More analysis is required to determine precisely what code must be duplicated in each processor to maximize local instruction fetches at low storage cost. And, as mentioned earlier, more analysis is required to determine the most cost effective use of microcode.

4. Performance of a Stand Alone Cm* System

L. Raskin

4.1. Introduction

Three application programs are described in this section along with some initial measurements of their operation on Cm*.

The three applications are:

1. Asynchronous Iterative methods for solution of PDE's.
2. Sorting (Quick Sort).
3. Set Partitioning Integer Programming.

Each application program is discussed below. The measurements obtained are presented and discussed.

4.2. Objectives

We have used these application programs as a workload for the Cm* system. This was done in order to measure the values and evaluate the importance of various attributes (like program/data locality) to:

1. Show the possibility of solving efficiently several problems on Cm* and thus the viability of such an architecture.
2. Make a theoretical comparison, using these parameters, between Cm* and other multiprocessor architectures (e.g., loosely coupled computer networks or tightly coupled structures such as C.mmp.) Measurements of these attributes will help in deriving and validating a performance model for Cm* like architectures.
3. Identify performance bottlenecks in the hardware system that will provide some insight and help in tuning and constructing such systems in the future.

4. Provide some quantitative criteria to decompose future applications for Cm* into parallel tasks.

5. Determine software (Operating System) and I/O system impact on these applications (e.g., in terms of overhead involved in synchronization, memory management). Help in tuning these systems.

6. Help to define classes of problems that are most suitable to run on Cm*

4.3. The Programs

4.3.1 Numerical Application - Partial Differential Equations.

4.3.1.1 The Problem

An example of an Asynchronous Iterative Method is the solution to Dirichlet's problem of Laplace's Partial Differential Equation (PDE) by the method of Finite Differences.

This program solves the PDE:

$$\frac{\partial^2 U(x,y)}{\partial x^2} + \frac{\partial^2 U(x,y)}{\partial y^2} = 0$$

on a rectangular grid of size M x N, where only the values at the outer edges of the grid are given.

The Finite Difference method transforms the problem into a set of linear equations: $A\underline{x} = \underline{b}$, where $\underline{x}$ is an MN vector of all the points in the grid, A is an MN x MN sparse matrix and $\underline{b}$ is an MN vector derived from the boundary conditions. This set of linear equations is derived from the new approximate values of the points (in each iteration) by averaging the values of the four adjacent neighbors of each point. The solution to this PDE is required in many application areas (e.g., Electro-Magnetic field, Hydrodynamics). Other PDE problems can be similarly solved using this method.

More details about Asynchronous Iterative Methods and their applications can be found in [BAUDET 76a, BAUDET 76b].

4.3.1.2 The Methods

Baudet [BAUDET 76b] gave a survey and developed several new methods for solving the above problem. Four of these methods were implemented and measured on Cm*. In all the methods, the computation is initially decomposed into P processes where P is equal to the number of processors available. Each process (and processor) iterates on a fixed subset of MN/P components out of the total MN components. The four methods are briefly discussed below:

Method 0: Jacobi's Method. In this method each processor retrieves its particular subset of the data from the global vector, $\underline{x}$, at the beginning of each iteration. New values are computed for the elements of $\underline{x}$ and then compared with their previous values. The elements are stored back into the global vector, $\underline{x}$, for other processors to use. This store operation is protected by a critical section. The processor then checks the error vector (computed by differencing the old and new values of the $\underline{x}$ vector). If the error vector is smaller than the pre-specified limit, the processor notifies the other processors that it has finished. If all of the other processors have finished their work the computation is complete. Otherwise the processor blocks awaiting the completion of the iteration by all the other processors before starting the next iteration.

Method 1: Asynchronous Jacobi Method (AJ). This method is the same as the Jacobi method (Method 0) except that each processor does not wait for the other processors to finish before starting on the next iteration.

Method 2: Asynchronous Gauss Seidel Method (AGS). This method is similar to AJ method (Method 1) except the processor uses the new values computed in its subset as soon as they are available (not the values known at the beginning of the iteration as in the previous two methods).

Method 4: Purely Asynchronous Method (PA). This method computes a new value of each component using the most recent values of all components by reading them directly from the global vector, $\underline{x}$, and writing the updated values directly back to the global vector (without any critical sections or synchronization). This last method is clearly the most efficient. It also uses less memory than the other methods. It uses critical sections rarely to inform the master process that the work has been finished. Almost linear speed up can be achieved theoretically with this method.

More discussion of the above methods and experimental results on C.mmp (using floating point) can be found in [BAUDET 1976 a,b]. Fixed point, single precision computation was used in the Cm* implementation.

Some features of this problem:

i. Extensive use of integer arithmetic operations.

ii. 4 different methods with different synchronization requirements.

iii. Can be compared with a C.mmp implementation.

iv. Almost linear speed up can be expected.

4.3.1.3 The Implementation

The grid size was chosen to be 21 x 24 points (i.e., a linear system of 504 elements). The boundary conditions were chosen to be all zeroes and the grid points initialized to one. The error bound was chosen to be 0.1 in all the following experiments. One processor, the *master* processor, initializes and starts the other *slave* processors, and prints the results when all have finished. Note that the *master* participates in the computation like any other (*slave*) processor. All the global variables are kept in the *master* processor's local memory area. Synchronization and mapping are achieved by using the simple Kmap micro-code.

4.3.1.4 The Results

The timing measurements and speed up factors for various memory reference patterns are given in Figures 1.1 to 1.8 and in Table 1. (The figures appear at the end of this section.)

Table 1: PDE, Memory Reference Patterns (using one Cm)

This table shows program execution times for various memory reference patterns expressed both in seconds and as percentages of the times taken when all references are to local memory.

Method 0

Memory reference pattern	Execution time	% Local execution time
All Local	362	100
All Mapped	954	263.5
Only Code Mapped	835.5	231
Only Stack Mapped	420	116
Only Local Variables Mapped	405	112
Only Global Variables Mapped	378	104.5

Method 1

Memory Reference Pattern	Execution Time	% Local execution time
All Local	355.5	100
All Mapped	948	267
Only Code Mapped	820	231
Only Stack Mapped	413	116
Only Local Variables Mapped	399	112
Only Global Variables Mapped	370	104

Method 2

Memory Reference Pattern	Execution Time	% Local execution time
All Local	181	100
All Mapped	478	264
Only Code Mapped	417	230
Only Stack Mapped	210	116
Only Local Variables Mapped	203.5	112
Only Global Variables Mapped	188	104

Method 4

Memory Reference Pattern	Execution Time	% Local execution time
All Local	165.5	100
All Mapped	433	261.5
Only Code Mapped	382	231
Only Stack Mapped	196	118.5
Only Local Variables Mapped	176.5	107
Only Global Variables Mapped	173	104.5

4.3.2 Sorting - Quick Sort

4.3.2.1 The Problem

This problem concerns the decomposition of the well known Quicksort algorithm [SINGLETON 69] into asynchronous parallel processes. The median for each sort pass was chosen as the median of the first, middle and last elements in the sublist. Each process is assigned to its own processor. (Hence the *process* and *processor* may be used interchangeable here.)

During a pass, each processor partitions its set of elements into two subsets: Elements larger than the median of the original set and elements smaller than the median. The processor then pushes the address and size of the smaller of the two subsets onto a stack shared by all the processors. (Making the smaller subset available to the other processors tends to put more work onto the shared stack in order to keep as many processors as possible, busy.) The processor proceeds to further partition the remaining (larger) subset. When the remaining subset cannot be partitioned further, the processor selects the next available subset from the shared stack.

Very simple assumptions about the algorithm (similar to the T = c N Log N for sorting using the sequential algorithm) give a theoretical sorting time of:

$$T = c\,N\,\{\,(K-M)/P + 2(1 - (1/2)^{M})\,\}$$

Where: N = # of elements to sort,
$K = \mathrm{Log}_2\,N$,
c = constant,
P = # of processors,
$M = \mathrm{Log}_2\,P$,

When the number of processors is much smaller than the number of items to be sorted almost linear speed-up can be achieved. The performanced degrades considerably when the number of processors is large (asymptotically to a constant speed of T = c Log N/2). See Stone [STONE 71] for a description of sorting methods that speed up as N/log N for large numbers of processors.

4.3.2.2 The Implementation

One processor, the *master* processor, initializes and starts the other processors. It makes the first partition and prints the results when the sort is complete. The *master* also participates in the sort like any other (*slave*) processor.

The stack, the vector of elements to be sorted, and the global variables are kept in the local memory of the *master* processor. All the experiments sort 18,000 elements, where each element is a 16 bit (2's complement) value.

Some features of this problem:

i. Extensive access to the shared data vector (which causes data contention).

ii. Extensive use of logical operations.

iii. Almost linear speed up when the number of elements in the data part is orders of magnitude larger than the number of processors.

4.3.2.3 The Results

The timing measurements and speed up factors for various memory reference patterns are given in Figures 2.1, 2.2 and Table 2. (The figures appear at the end of the section.)

Table 2. Quicksort, Memory Reference Patterns (using one Cm)

This table shows program execution times for various memory reference patterns expressed both in seconds and as percentages of the times taken when all references are to local memory.

Memory Reference Pattern	Execution time	% Local execution time
All Local	25.54	100
All Mapped	70.0	274
Only Code Mapped	58.4	229
Only Stack Mapped	30.3	118.5
Only Local Variables Mapped	28.3	111
Only Global Variables Mapped	31.2	122

4.3.3 Searching - Set Partitioning Integer Programming

4.3.3.1 The Problem

The particular integer programming considered here is one of the most practical and applicable methods. It is used, for example in airline crew scheduling [BALAS 76].

This problem is typically solved with an enumeration algorithm, by searching (N-ary tree search) in a large, relatively sparse, binary matrix - typically on the order of hundreds x thousands - for a minimum cost solution.

The set partitioning problem is to solve:

$$\min \{ \underline{c}.\underline{x} \mid A\underline{x} = \underline{e}, x_j = 0 \text{ or } 1 \text{ for } 0 \leq j \leq N \}$$

Where: $A = M \times N$ binary matrix.
$\underline{c}$ = N *vector*
$\underline{e}$ = (1.....1) M vector

As an example of this method, consider the airline crew scheduling problem. The rows of the A matrix correspond to a set of flight legs (from city A to city B, in time T) to be covered during a specified period and the columns of A correspond to a possible sequence of tours of flight legs done by one crew, $\underline{c}$ is the vector of associated cost of each tour. A possible solution includes a set of tours that satisfy all the flight legs (one and only one crew makes a flight leg). We are looking for the solution with the lowest cost.

Some features of this program:

i. It uses binary data types, manipulates a large matrix in a relatively small address space.

ii. Extensive use is made of both arithmetic & logic functions.

iii. There is a theoretical possibility of nearly linear speed up.

iv. The application is relatively complex.

4.3.3.2 The Implementation

As in the previous applications, one processor - the *master* - initializes, creates the array according to user's specification and puts enough initial possible search path solutions in a global stack, from which all the processors pick their work. The criteria was (arbitrarily chosen) to put more than 10 x P path solutions into the stack (where P = # of processors) - so that the work will be more evenely distributed between the processors and all will be occupied for a large percentage of the time.

To enhance pruning in the search, a global variable contains the cost of the best solution found so far by any of the processors - and all compare their current cost value to it and begin to track back in the search when that global cost is lower.

4.3.3.3 The Results

Five different cases where arbitrarily chosen as test cases.

-	I	J	Seed	Density	# Solutions
Case 1	10	100	1	0.1	3
Case 2	10	100	2	0.1	5
Case 3	10	100	3	0.1	5
Case 4	50	500	1	0.2	0
Case 5	17	60	1	0.1	1

Where:

I = # of rows in the A matrix
J = # of columns in the A matrix
Seed = Initial seed number for a random # generator to generate the matrix
Density = Density (ratio of ones and zeroes) in the array
Solutions = # of different solutions found by one processor

The timing and speed up factor results for the 5 different cases are given in Figures 3.1 to 3.2 and in Table 3. (The figures appear at the end of the section.)

Table 3: Integer Programming, Memory Reference Patterns (one Cm)

This table shows program execution times for various memory reference patterns expressed both in seconds and as percentages of the times taken when all references are to local memory.

Case 1

Memory reference Pattern	Execution Time	% Local Execution Time
All Local	79.1	100
All Mapped	215.1	272
Only Code Mapped	176.8	223.5
Only Stack Mapped	113.9	143
Only Local Variables Mapped	85.8	108.5
Only Global Variables Mapped	81.1	102.5

Case 2

Memory reference Pattern	Execution Time	% Local Execution Time
All Local	19.5	100
All Mapped	53.0	272
Only Code Mapped	43.6	223.5
Only Stack Mapped	27.3	140
Only Local Variables Mapped	21.1	108.2
Only Global Variables Mapped	20.0	102.5

Case 4

Memory reference Pattern	Execution Time	% Local Execution Time
All Local	204.4	100
All Mapped	546.1	267
Only Code Mapped	455.5	223
Only Stack Mapped	277.4	136
Only Local Variables Mapped	217.5	106.5
Only Global Variables Mapped	208.3	102

4.4. Initial Results

4.4.1 Access times in the Cm* system

The following summarizes the results presented in Tables 1 to 3. The figure given is the ratio of total execution time (for various memory reference patterns) to the total execution time when all memory references are to local memory.

1. When all references are mapped, the ratio is between 2.6 and 2.75.

2. When code is mapped and everything else is local, the ratio is between 2.2 and 2.3. Hence it is very important for code to be in a Cm's local memory.

3. When the Cm's stack is mapped, the ratio is between 1.16 and 1.185 for the PDE and QuickSort applications and 1.4 for the Integer Programming application. The latter application consists of a large number of small routines, the execution of which causes frequent stack accesses to perform the call/return sequence.

4. When global data is mapped, the ratio varies between 1.02, when global data accesses are infrequent, and 1.15 when accesses are relatively frequent. These particular figures are encouraging since large shared data structures may be located anywhere in the system without significant performance degradation.

5. When Own (local) data is mapped, the ratio is between 1.07 and 1.12.

4.4.2 Throughput of Cm* Buses and Components.

1. When all processors share both code and data from a single memory, the graphs indicate that performance cannot be improved by using more than 3 or 4 processors. This limitation is caused by memory contention.

2. The graphs showing Cms making references which are mapped back to their local memory ("Mapped to Self") indicate that the Kmap saturates when 6 or 7 processors are simultaneously active in this mode.

4.4.3 Hit and references ratios

The rate of mapped memory references from a Cm, for which *all* references were mapped, was measured. The reference rates to Code, Stack, Own (local) and Global data were also measured. The percentages of the total reference rate represented by the above reference types are tabulated below. (The measurements were made using a combination of the Map Bus Monitor and a frequency counter.)

a. PDE

-	method 0	method 1	method 2	method 4
Code	80.8%	80%	80.5%	82%
Stack	10%	10%	10%	11.5%
Owns	6.8%	7%	7%	4%
Globals	2.5%	3%	2.5%	2.5%

b. Quick sort

Code	71%
Stack	12.5%
Owns	6.5%
Globals	9.5%

c. Integer programming

-	case 1	case 2	case 3	case 4	case 5
Code	71.3%	70.3%	71.1%	72.8%	71.5%
Stack	23%	24%	25.4%	22.3%	23.6%
Owns	4.75%	4.6%	4.15%	3.75%	3.85%
Globals	1.1%	1.1%	1.2%	1.1%	1.1%

The hit ratios are therefore on the order of 97.5% in the PDE program, 90.5% in the Quicksort program and 99% in the Integer Programming program.

4.4.4 Utilization

4.4.4.1 System Components

Figures 4.1 to 4.4 show the different aspects of the system's component utilization while running the PDE programs. The Kmap executes a 2 micro-instructon loop (called the "idle loop") whenever there is no useful work for it to do. (This is used to show the utilization of the Kmap). The "Slocal busy loop" is 7 micro-instructions long and is used to show the amount of contention for Slocals and memories. This loop is executed when the Kmap tries to read from, or write into, a memory but the Slocal is still busy executing a previous reference. A successful reference to an Slocal takes 4 micro-instructions.

The Kmap micro instruction time is about 157 nanosec.

It is interesting to note that the maximum Pmap utilization when the hit ratio is 0% and all references are mapped back to the Cm is only 35% (the rest of the time is spent in the idle loop).

4.4.4.2 Number of Cms supported by a Kmap

The measurements shown in Fig 4.2 (all references mapped) shows that, due to Kmap contention, there is a degradation of about 10% in the time to execute a remote (mapped) reference inside the cluster when about 400,000 references/sec. are made to the Kmap.

Consider, for example, the PDE program. The average number of mapped references from one Cm, when only global variables are mapped, is about 6,500 references/sec. This means that the Kmap can support 400,000/6,500 or about 60 Cm's with only 10% degradation in the mapped reference time. With a 90% hit ratio (10% mapped references), as observed in the Quicksort program, the Kmap can support about 20 processors with a 10% degradation in the mapped reference time.

4.4.5 Total local memory references per second and MIPS.

Based on the three programs measured, the local memory reference rate was as follows:

PDE - Time between successive memory references = 3.7 microseconds. Reference rate = 270 KHZ.

Quick Sort - Time between successive memory references = 3.33 microseconds. Reference rate = 300 KHZ.

Integer Programming - Time betwen successive memory references = 3.51 microseconds. Reference rate = 285 KHZ.

The average number of memory references per instruction was measured to be:

QUICKSORT	- 1.75
PDE	- 1.45
INTEGER	- 1.75

For HARPY the reference rates are in the 1.85 - 1.9 range. (See section 5.)

These numbers are considerably lower than the corresponding numbers (2.1 to 2.3) measured on a large sample (about 8 million instructions) of from 20 Fortran, Cobol and system programs [MARATHE 77]. A possible explanation for this may be that the Cm* applications were compiled by a good optimizing compiler.

The average number of memory references per instruction in the measured programs was about 1.7. From this we get that the maximum potential instruction rate of a Cm executing a program in its local memory is on the order of 0.17 MIPS (Millions of Instructions Per Second). This may be extrapolated to 1.7 MIPS for a 10 processor system.

4.4.6 Memory Contention

As seen in Fig. 4.4 (Slocal busy loop count) the time added to a reference by each Slocal busy loop count is about 2.26 microseconds (1.1 microseconds in the Pmap and 1.16 microseconds in the Kbus and Map bus).

4.4.6.1 PDE

The number of busy loop references in the PDE program increases from 0 to 4200 in Method 0 and to 9600 in Method 4 - with 0 to 8 Cm's all running local code (and shared data). This means that, as expected, the degradation due to memory contention in this high hit ratio application is negligible. The memory contention adds only 0.5% - 1% to the utilization of the Pmap. (The Pmap spends 2% - 3% of its time doing useful work, and sits in its idle loop for the remaining 96% - 97% of the time.)

From the above results, the performance degradation due to memory contention (and hence slower mapping) was calculated to be in the range: 0.1% to 0.25%.

4.4.6.2 Quick Sort

This program is difficult to measure as it has a short execution time and changes its demands upon the Kmap during execution. Taking peak Kmap activity with 8 *processors* (local code):

Slocal busy loop	100,000 counts (11.0% of Pmap time)
Idle loop	2,500,000 counts (78.5% of Pmap time)
Useful mapping references	150,000 counts (10.5% of Pmap time)

From the above results, the performance degradation due to memory contention was calculated to be 2.7%. Map bus contention was ignored in the above calculations.

4.4.7 Frequency of Lock operations

The total number of lock operations per second ranged, in PDE Method 0, from 10 in a 2 Cm system to 3,400 for the 8 Cm system. For Quick Sort the range is 1,500 - 14,000. These are algorithm dependent numbers and do not have much influence on total execution time due to the fast synchronization operation in the simple Kmap microcode.

4.4.8 Estimate of execution times using multi cluster Cm* configuration

The number of mapped references were measured from Cm #0 - Cm #3 when the code was local and the global data was in Cm #13.

	PDE		Quick Sort
	Method 0	Method 4	
Reference Counts			
Total (Mapped) Cm's 0-3	22000	27000	90000
Total for 1 Cm	257200	256000	256000
Mapped for 1 Cm	5500	6750	22000
Average Memory Reference Times			
Local (Microseconds)	3.7	3.7	3.3
Mapped Intracluster	9.44	9.47	9.16
Mapped Intercluster (est.)	26.4	26.4	26.4
Reference ratio (see note)	1.095	1.115	1.39

Note: The intercluster/intracluster reference ratio was calculated by multiplying the number of local and mapped references by their respective execution times in the intercluster and intracluster cases, then dividing the two resultant numbers.

The references ratios show that the PDE method 0 algorithm (which waits for the slowest processor to finish) experiences a slow down of about 9.5%, but the slow down on method 4 and in Quicksort should be much better than the 11.5% and 39% predicted slow down in execution times - due to different algorithms (difficult to predict).

4.4.9 Some useful numbers

Some rounded, useful numbers from this evaluation of Cm* include:

1 Cm (i.e. single LSI-11 processor) - 0.17 MIPS

Reference saturation rate of shared memory bus - 270 KHz (3.7 microsec per reference).

Reference saturation rate of Kmap (including the map bus) - 552 KHZ

Saturation of map bus transactions (i.e read or Read-Modify-Write between 2 Cm's needs three transactions and write requires two transactions) - 1.7 MHZ

4.5. Measurements of Inter Cluster (Linc) communication

4.5.1 Program execution

Figs. 5.1 and 5.2 show the initial results from a multi-cluster Cm* configuration (as shown in figure 1 of the introduction).

The program used for that evaluation was the PDE program (methods 0 and 1). The Cm* configuration was a 3 Cluster system (4 - 4 - 2 Cm's in each) and the program was measured while using only two Clusters with equal number of Cm's in each.

Fig 5.1 shows that, with almost all the memory references local to each Cm (only globals mapped either inter-cluster or intra-cluster depending on the location of the Cm in the system), the increase in the execution time between using the Linc in a two Cluster configuration, or executing with all Cm's configured into the same Cluster - is only 5.5% to 12%.

Although the multi-cluster performance of Figure 5.1 is not much worse than the single Cluster configuration, it does raise the issue of when, if ever, it makes sense to consider multicluster configurations. With only 10 Cm's in the initial configuration we did not observe any practical situation in which multicluster Cm* configurations were superior to single Cluster. However, it is obvious that in larger Cm* systems a single Map bus and Kmap will become a bottleneck. Figure 5.2 shows a case, albeit an artificial one, in which a multicluster system is better than a single Cluster configuration.

Fig 5.2 shows the results with all memory references within a Cluster mapped back to the originating Cm and only globals shared across Clusters. Starting in 4 processors, a better execution time is achieved by using the Linc in a two Cluster (2 Cm's in each) configuration than by using one (big - 4 Cm's) Cluster - which increases to about 36% speed-up with 8 Cm's participating.

This is due to the high contention (and saturation) of the Kmap in the one Cluster (0% hit ratio) configuration while in the two Cluster case the work is divided between the two Kmaps participating with only a small percentage of the references (globals) being executed via the Linc (which does not significantly degrade the overall performance). This simulates a case where, in some special circumstances, using the Inter-Cluster bus relieves contention (and saturation) in a one Cluster - many Cm's - configuration.

A large Cm* system is necessary to test the utility of multicluster configurations for running practical applications.

4.5.2 Contention

Figs. 5.3 and 5.4 show the results of executing a simple, one instruction loop (L: mov #L, R7), test case through the Linc. The system was in a two Cluster (8 - 2 Cm's in each) configuration, with all references initiated in the source (8 Cm's) Cluster reading from memory in the other Cluster.

Fig 5.3 shows the change in reference rate in the source Cluster with an increasing number of processors participating. The Figure shows different memory access methods:

i. Sharing the same memory in the destination Cluster, or alternating between the memories of two Cm's.

ii. Using a single Linc, or alternating between two Lincs (ports).

Fig 5.4 shows the utilization of the source Pmap while executing the above cases. Utilization was obtained by counting the rate of Kmap Idle Loop iterations.

From these graphs a few observations and parameters can be deduced:

i. The time to execute an Inter-Cluster reference (without contention in the system) is 26.4 micro second.

ii. With only one processor executing, the source Pmap is busy only 13.2% of the time - I.e about 3.5 micro second of the Inter-Cluster reference time is spent in the source Pmap (the same as received from a theoretical calculation based on the number of microinstructions executed). Similar measurements of the destination Cluster shows it being busy 10.3% or about 2.7 microseconds per reference.

iii. From Figure 5.3, we see that a single Linc has a bandwidth of about 200,000 references per second. When multiple Lincs are used, the source Pmap becomes the bottleneck, and we do not reach saturation using 8 Cm's. Extrapolating from the utilization of the Pmap in Figure 5.4 (and from the 3.5 microseconds execution time per reference for the source Pmap) the Inter-Cluster saturation rate is estimated to be about 287,000 references/second.

iv. The anomalous behavior of some of the graphs in Fig 5.3 and 5.4, when severe contention occurs in the 7 to 8 Cm!s region, is not well understood yet and needs further investigation.

4.6. Conclusions

In this chapter we have studied the performance of the Cm* system using three programs. These programs were drawn from different application areas. A number of conclusions may be drawn from the measurements presented here; The most important are:

i. No single bottleneck component was discovered in the initial Cm* system. Furthermore, the results show that there is a potential for expanding the system without a significant system performance degradation.

ii. The differences in the memory reference time which is a function of the hierarchy in the Cm* memory structure (i.e local memory, intercluster or intracluster references) does not degrade significantly the efficiency of the system for solving the above problems. This is due to the high local memory hit ratio experienced by these programs - about 90% in the Quicksort program and approx. 97.5% in the PDE and Integer Programming problems.

iii. In such high hit ratio applications, it was shown that localizing the code, stack area, and local variables (in that order) is very important. The global data can be placed anywhere in the system (even outside the Cluster) without much degradation in the application's execution time.

iv. A larger Cm* system along with a larger, more complex, application program is needed to continue evaluation of the system, exercise it's full potential and explore it's limitations.

4.7. References

[BALAS 76] Balas E. and Padberg M., "Set Partitioning - A Survey" SIAM Review, Vol 18, No 4, Oct. 1976.

[BAUDET 76a] Baudet G. "Thesis Proposal", Comp. Science Dept., CMU, April 1976.

[BAUDET 76b] Baudet G. "Asynchronous Iterative Methods For Multiprocessors", Comp. Science Dept., CMU, TR, November 1976.

[MARATHE 77] Marathe M. Ph.d Thesis, Comp. Science Dept. CMU, (in preparation).

[SINGLETON 69] Singleton R.C. "Algorithm 347" CACM, Vol 13, No. 3, March 1969.

[STONE 71] Stone H.S. "Parallel Processing With The Perfect Shuffle", IEEE Trans. on Computers, Vol. C-20, No. 2, Feb. 1971.

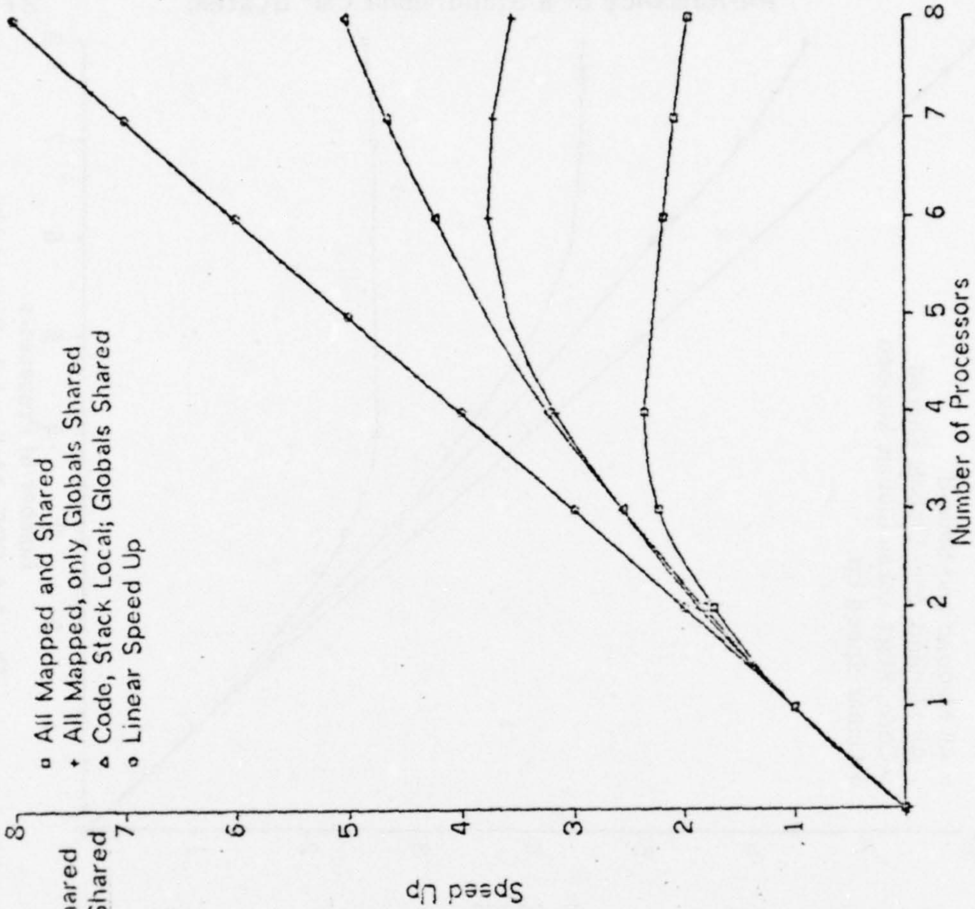


Fig 1.2 PDE, Method 0. Speed Up

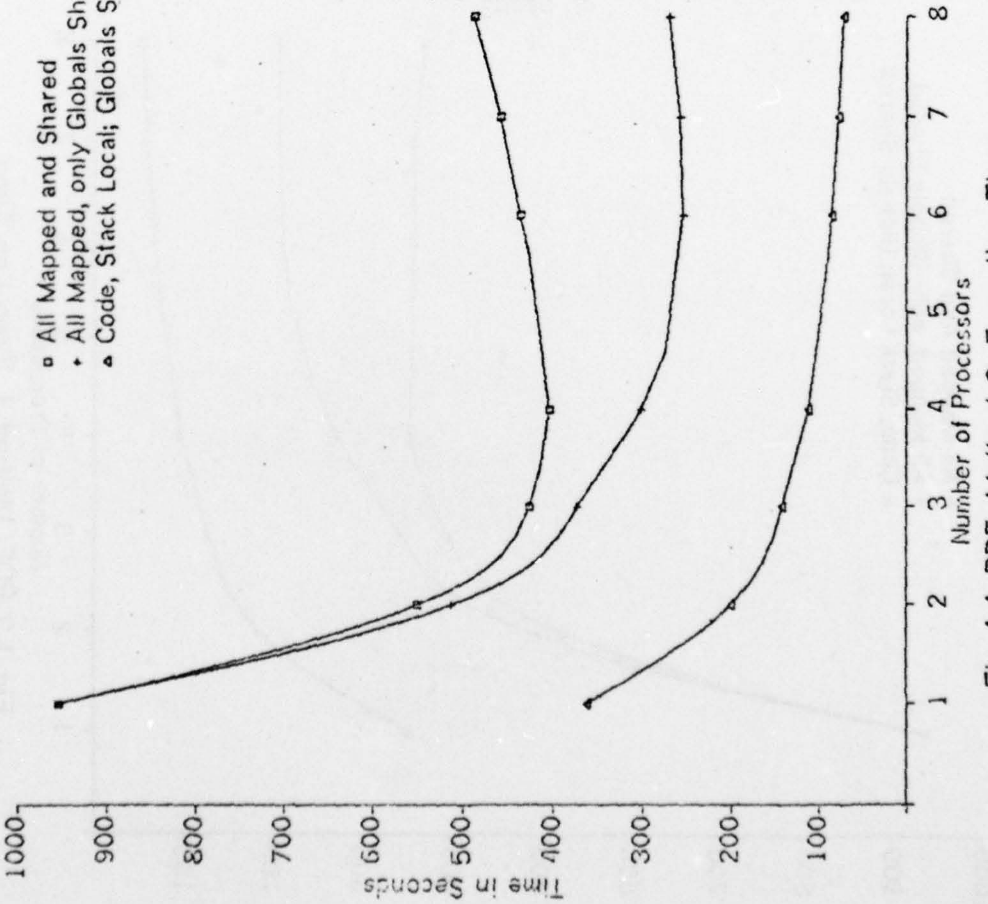


Fig 1.1 PDE, Method 0. Excution Time

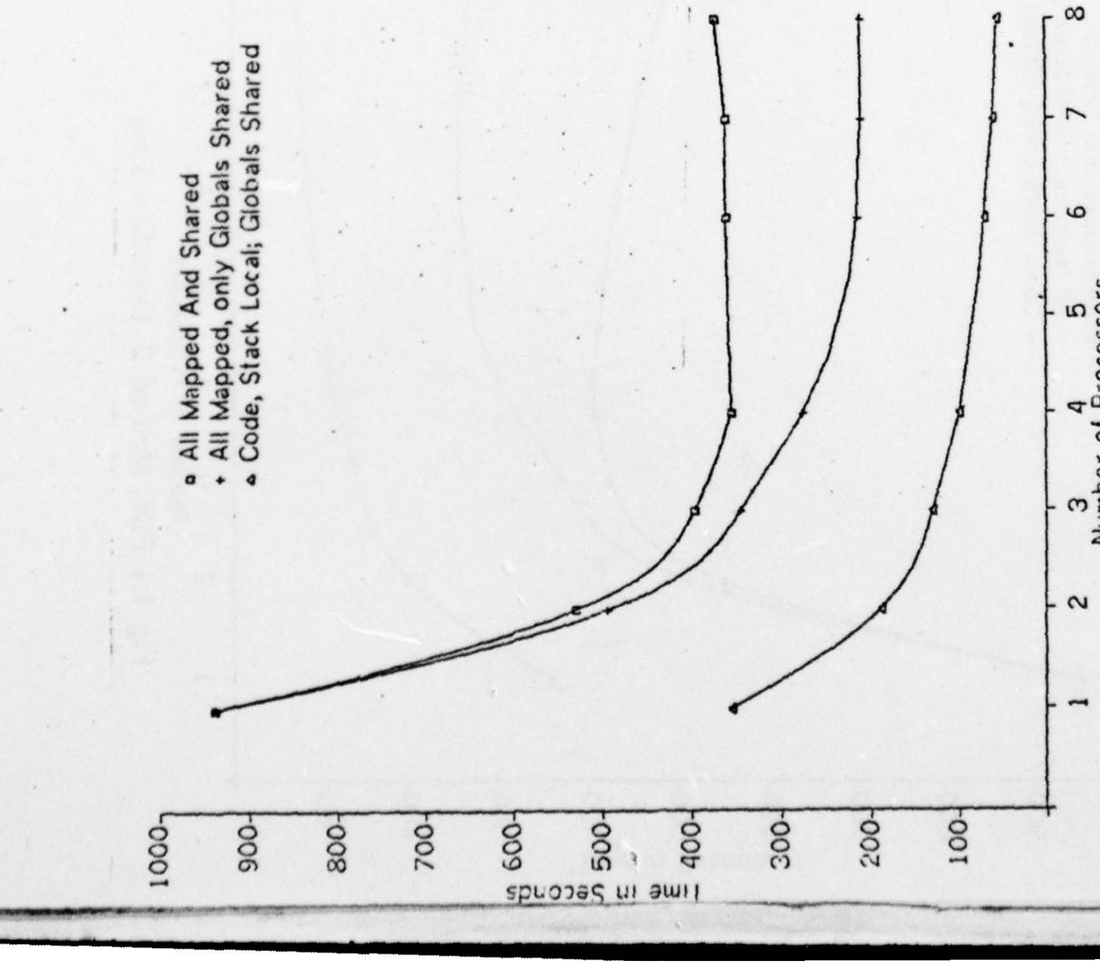


Fig 1.3 PDE, Method 1. Execution Time

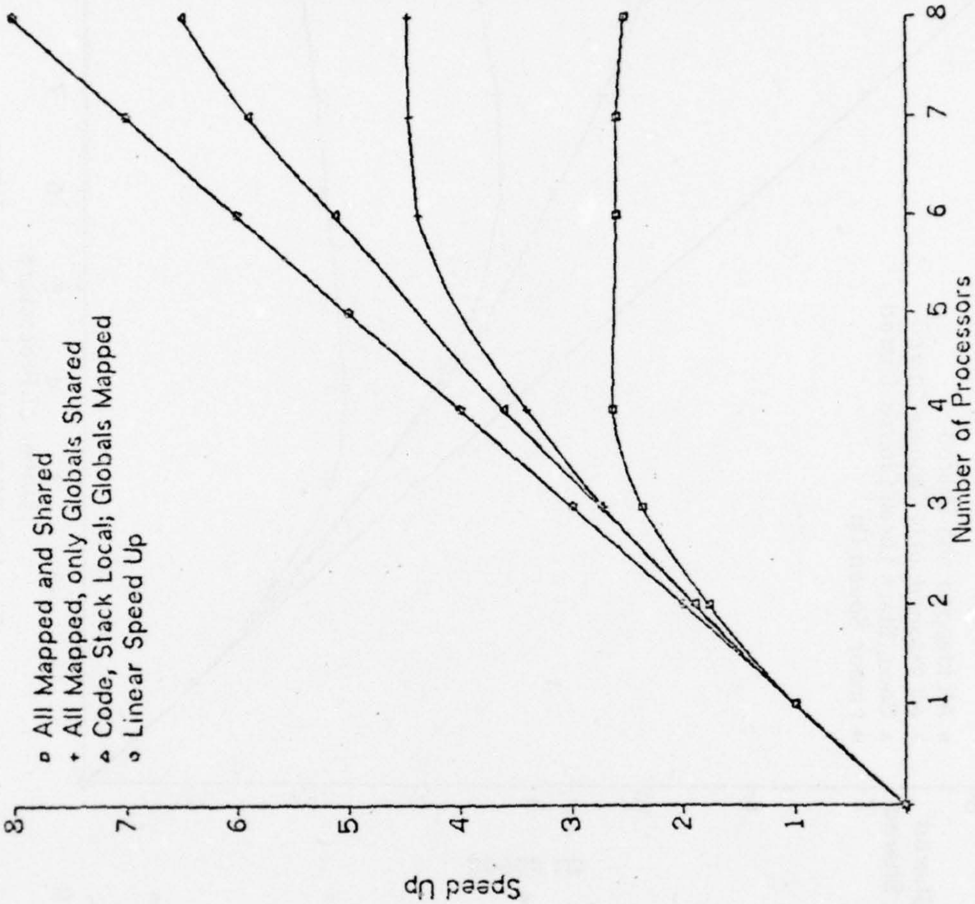


Fig 1.4 PDE, Method 1. Speed Up

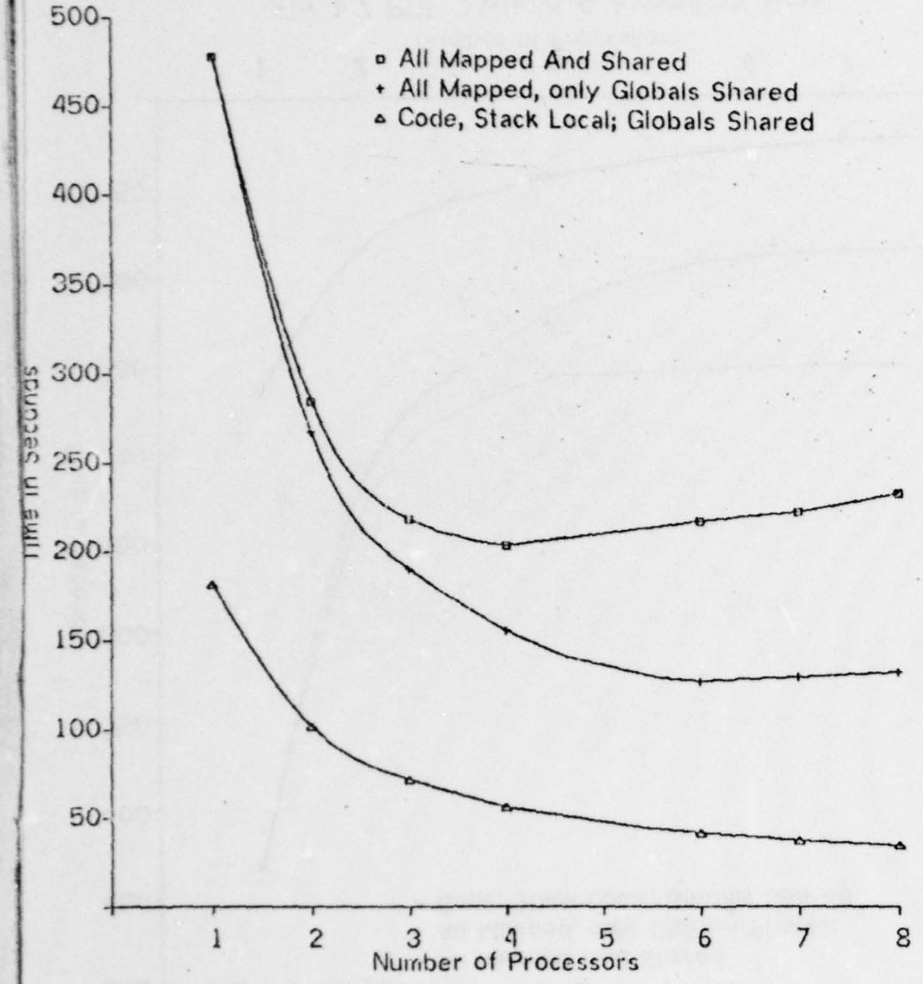


Fig 1.5 PDE, Method 2. Execution Time

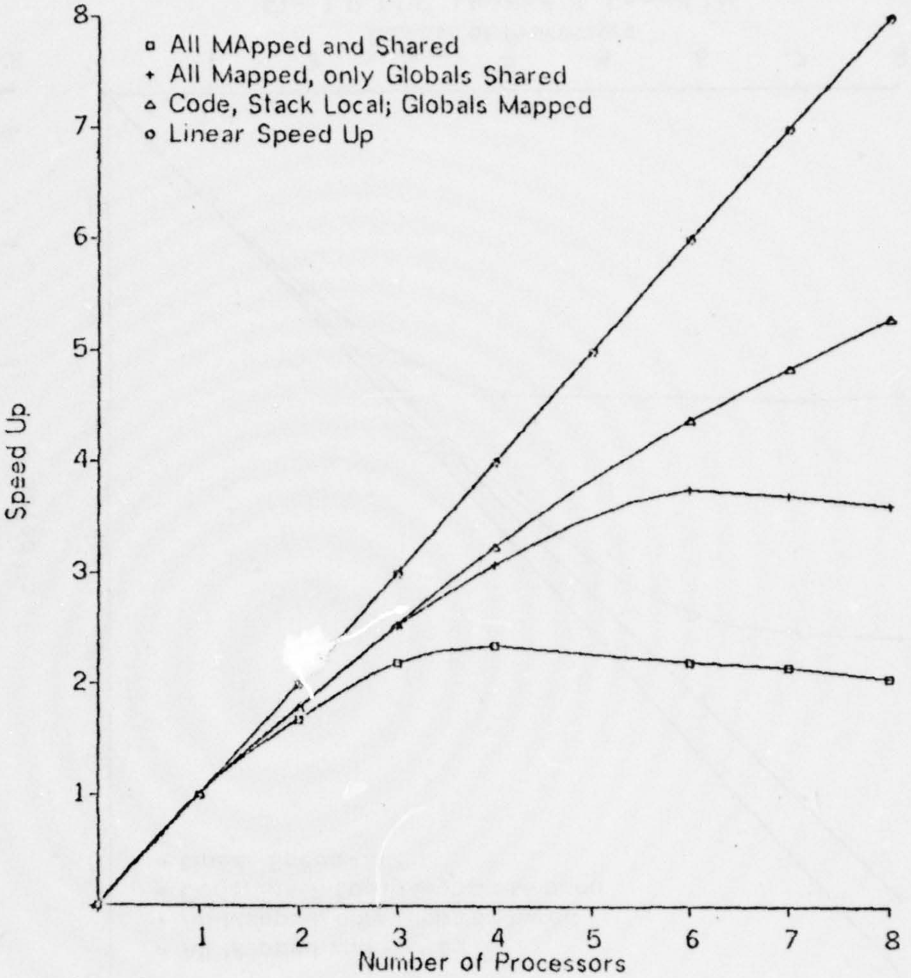


Fig 1.6 PDE, Method 2. Speed Up

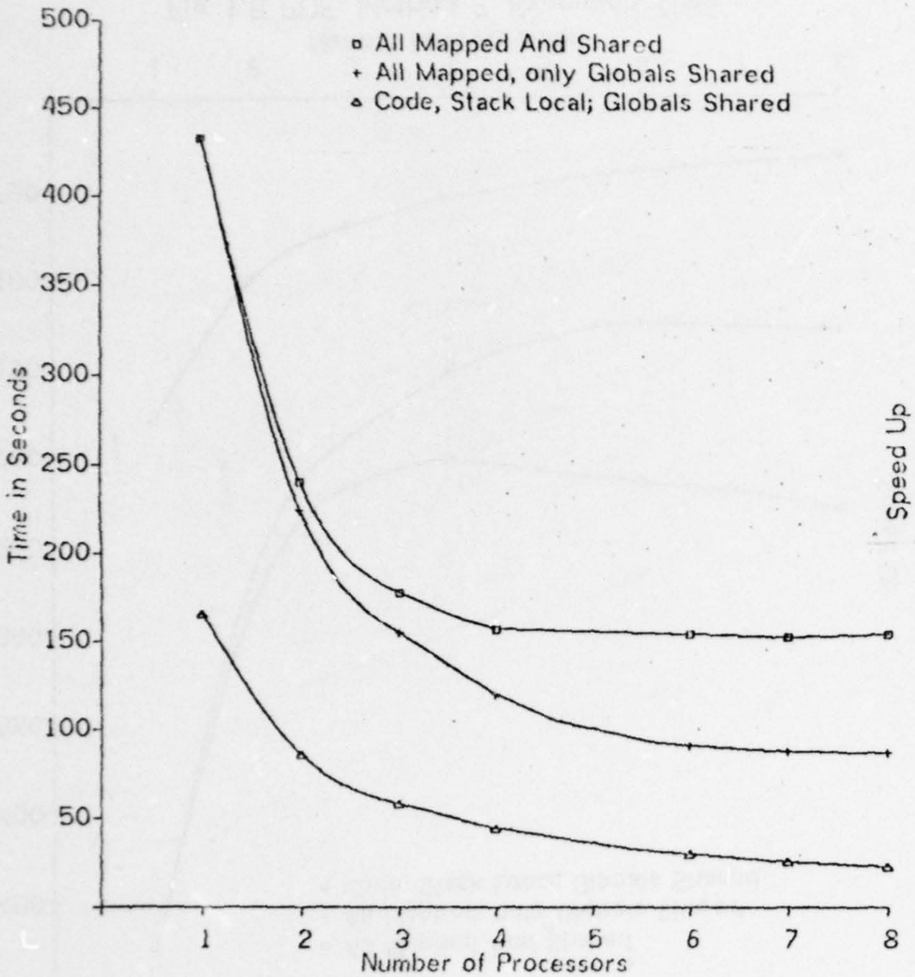


Fig 1.7 PDE, Method 4. Execution Time

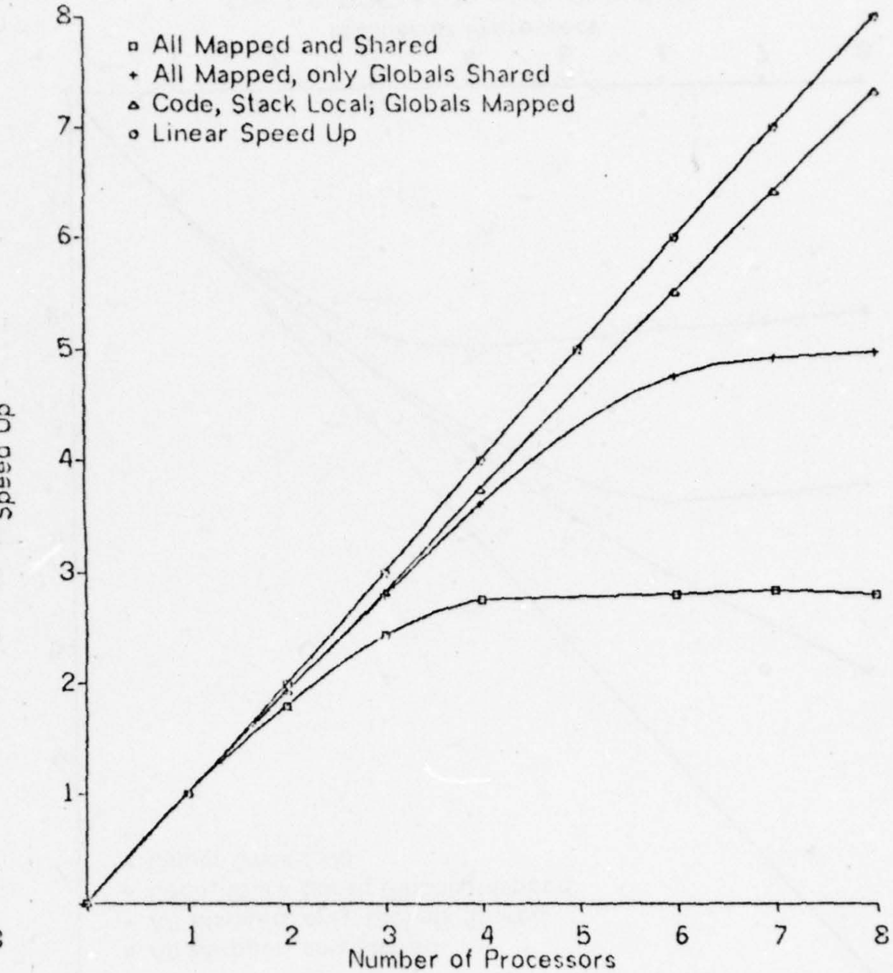


Fig 1.8 PDE, Method 4. Speed Up

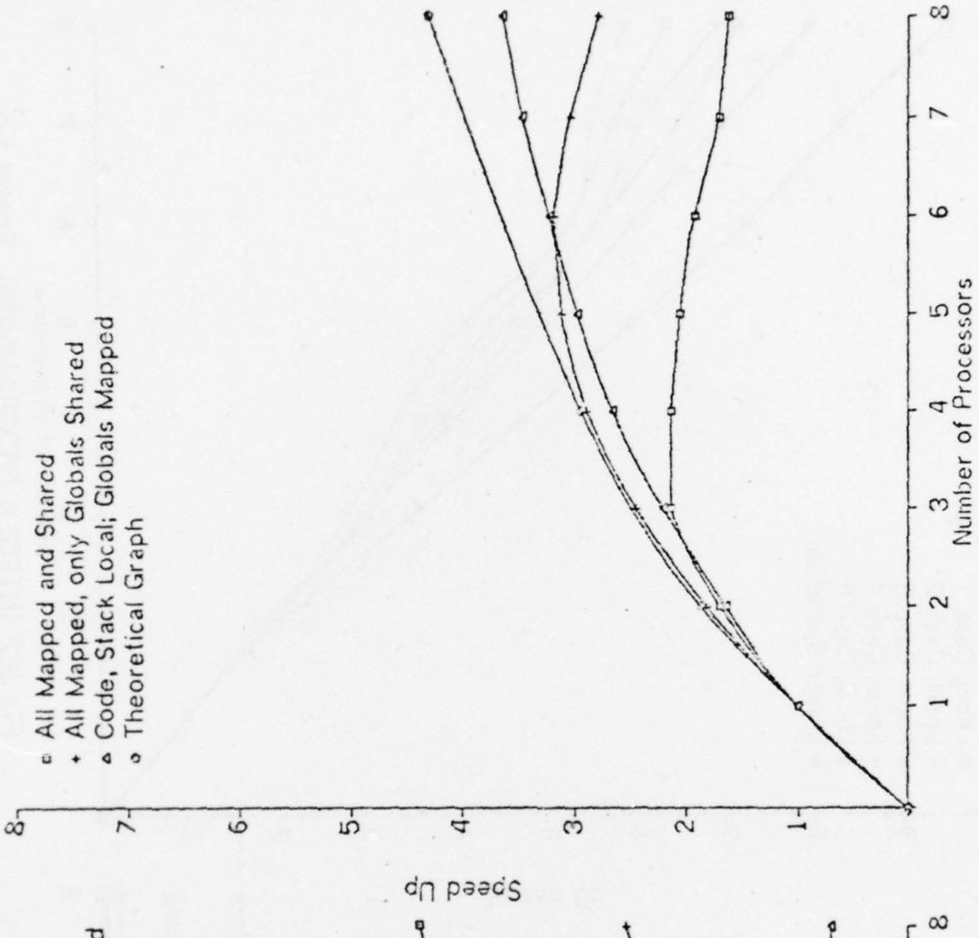


Fig 2.2 Quick Sort. Speed Up

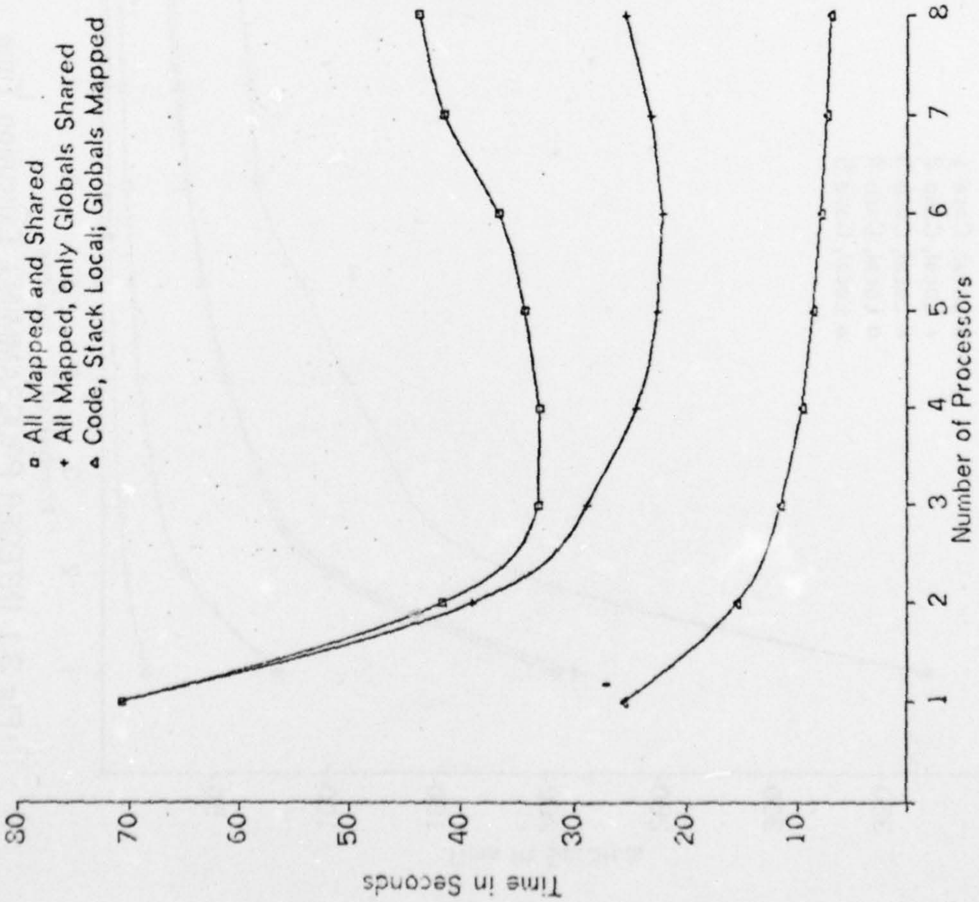


Fig 2.1 Quick Sort. Execution Time

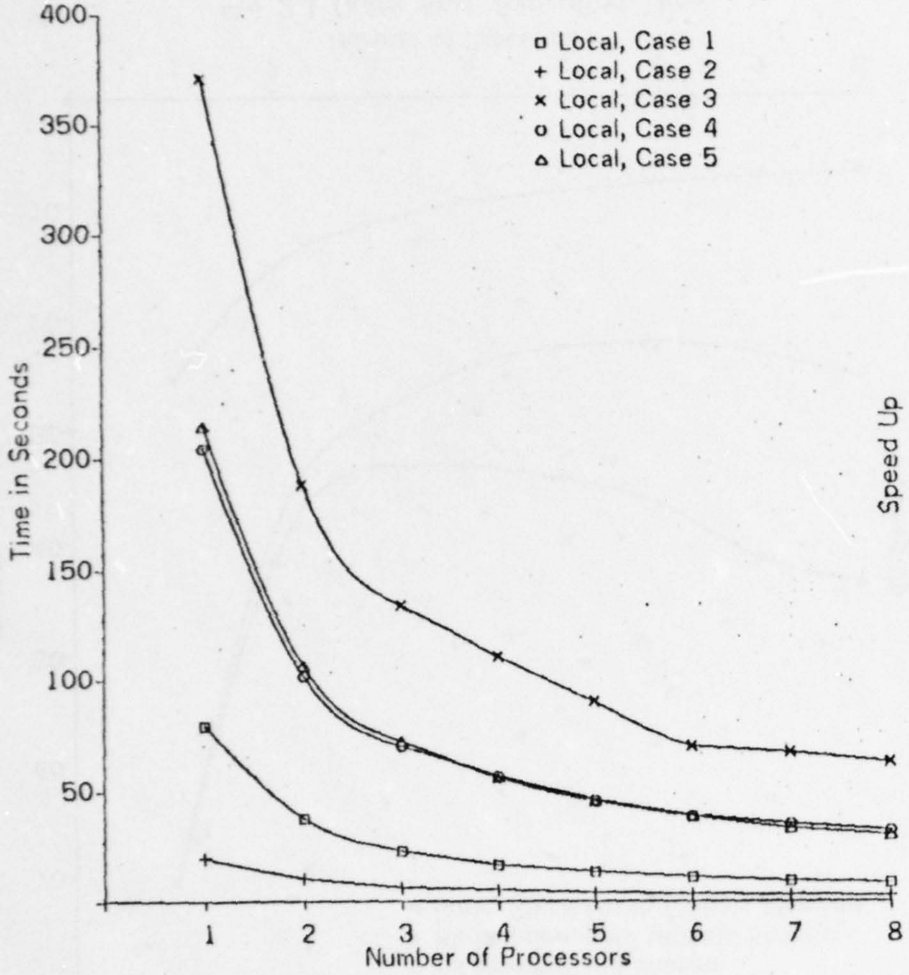


Fig 3.1 INTEGER PROGRAMMING, Execution Time

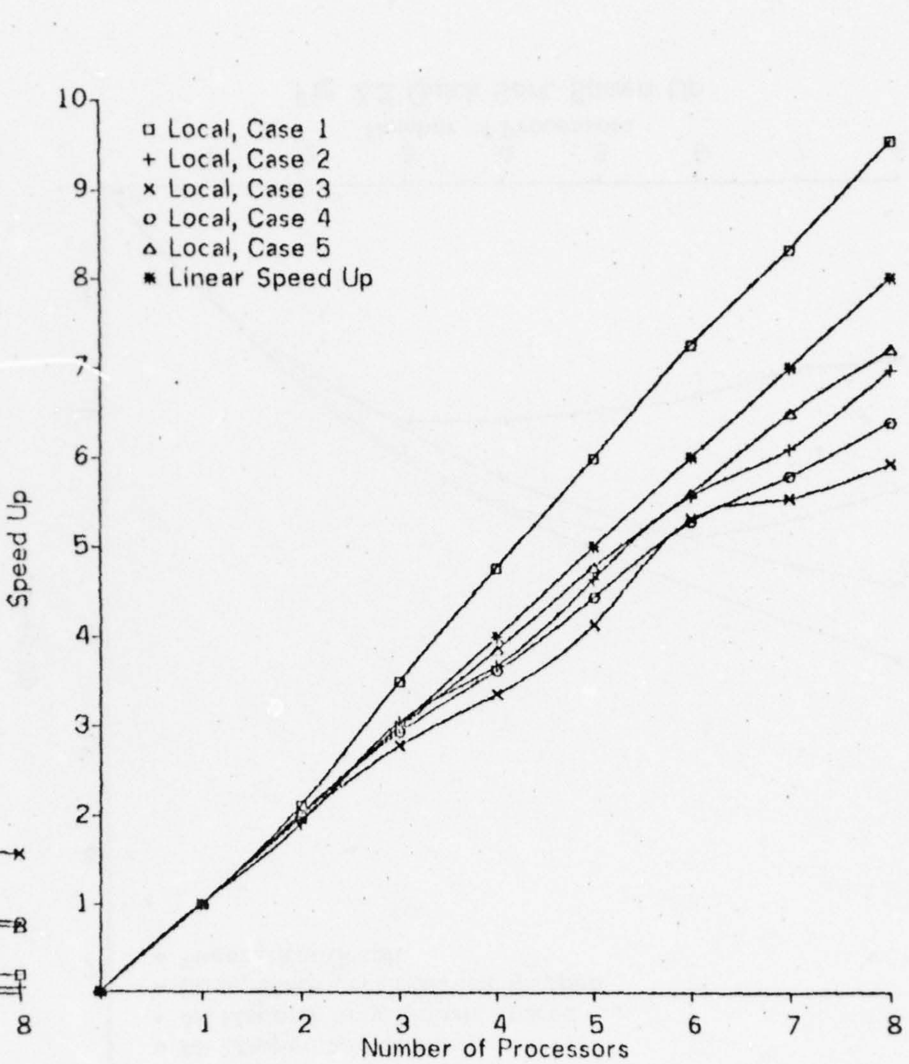


Fig 3.2 INTEGER PROGRAMMING Speed Up

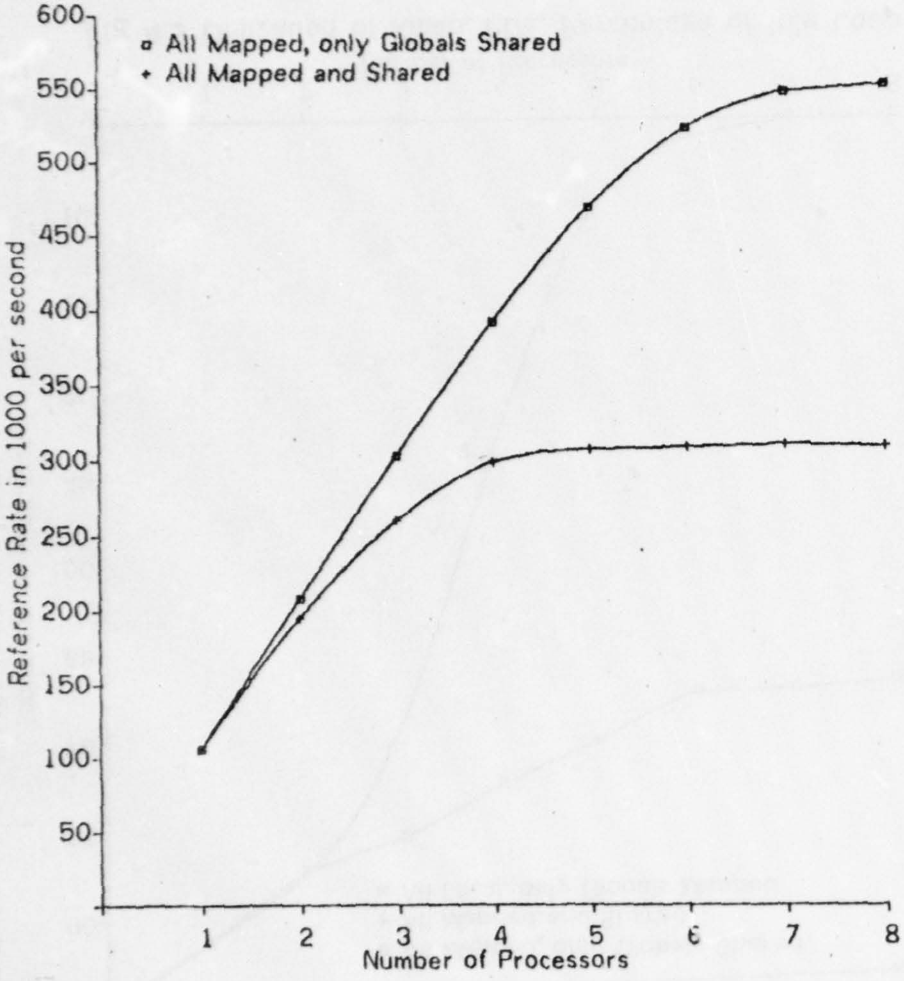


Fig 4.1 Utilization of Buses,

PDE- Total Reference Rate to K.map

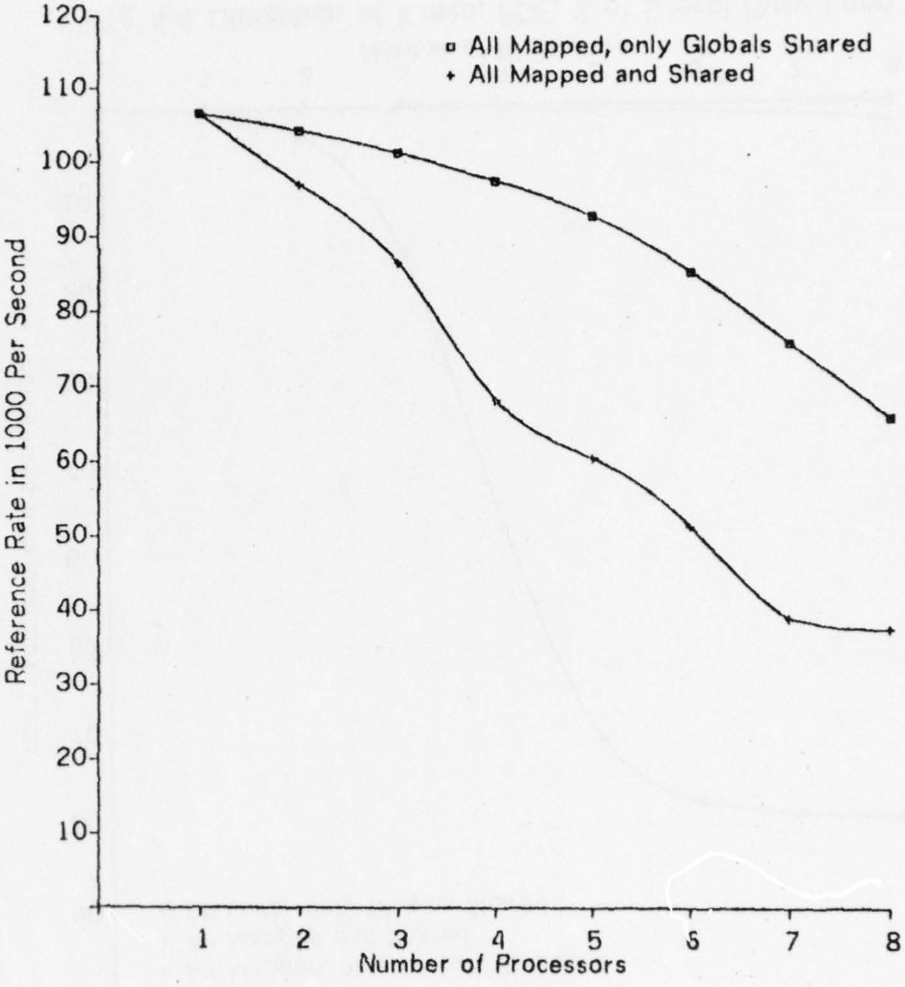


Fig 4.2 Utilization of Processors,

PDE- Reference Rate from one Cm

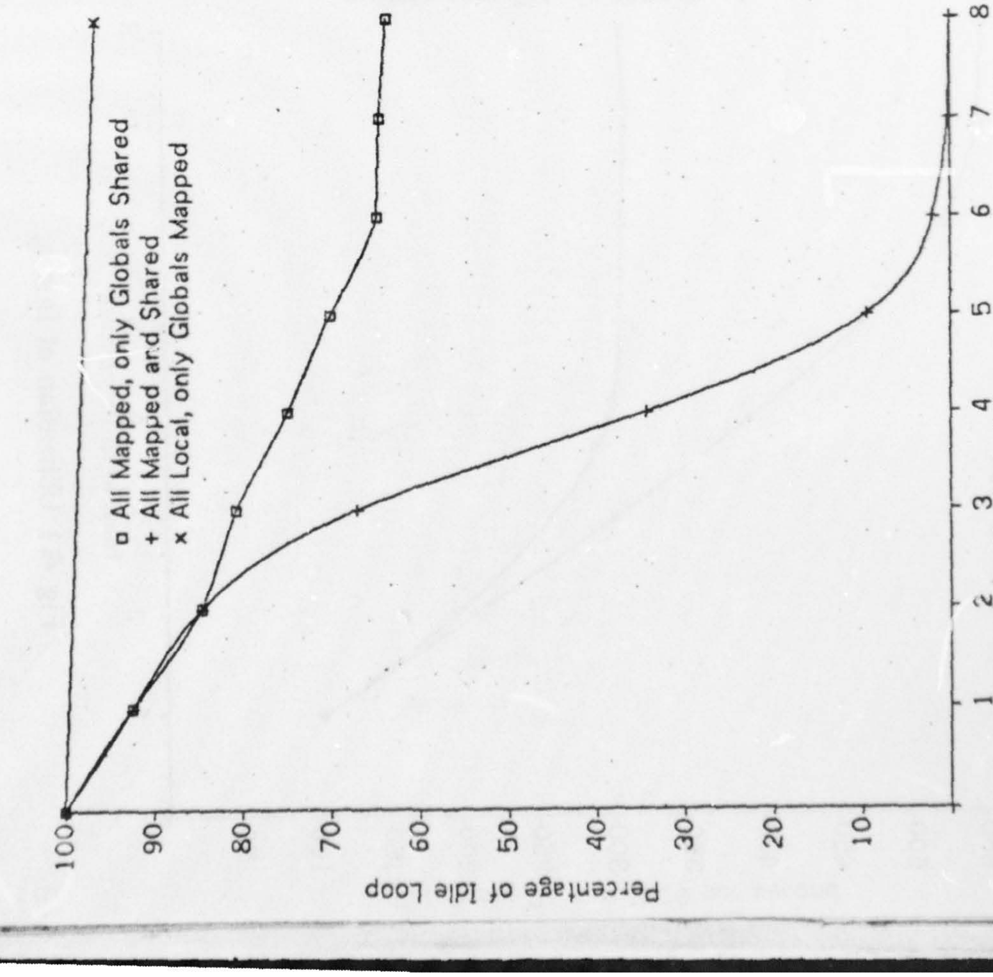


Fig 4.3 Utilization of Kmap, PDE, Percentage of Idle Loop

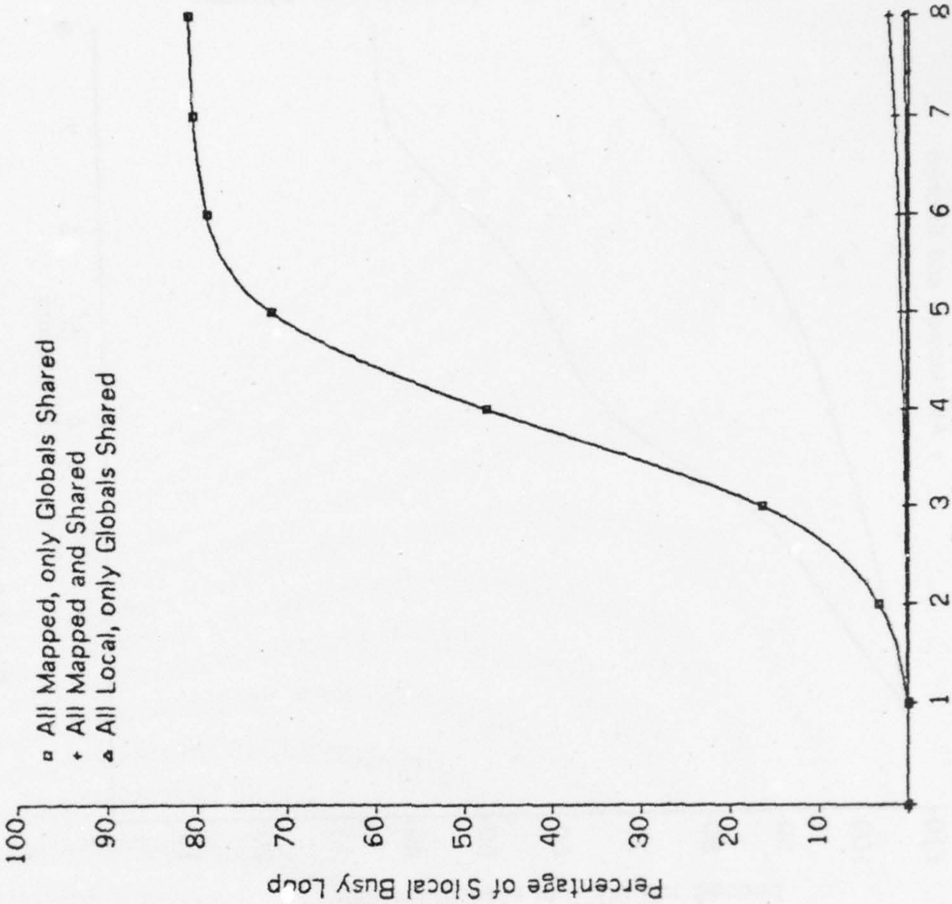


Fig 4.4 Utilization of K.map, PDE, % of S.local Busy Loop

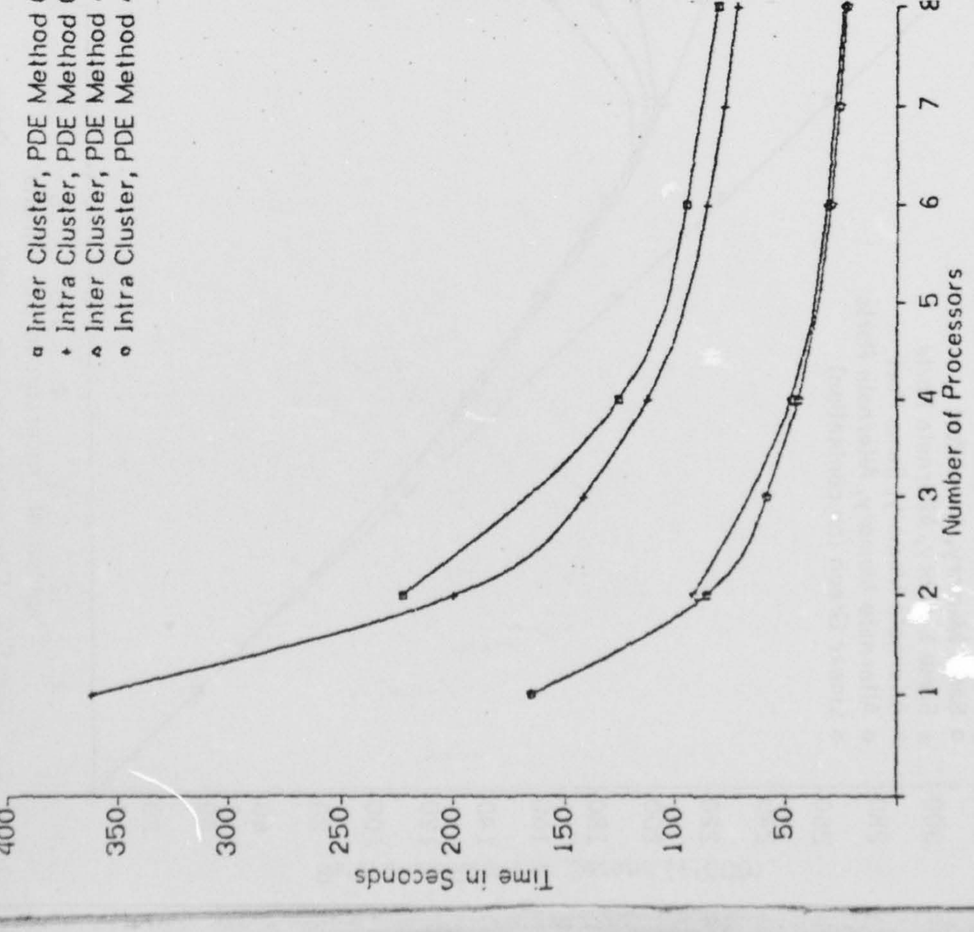


Fig 5.1 Intra & Inter Cluster Execution Time (local code)

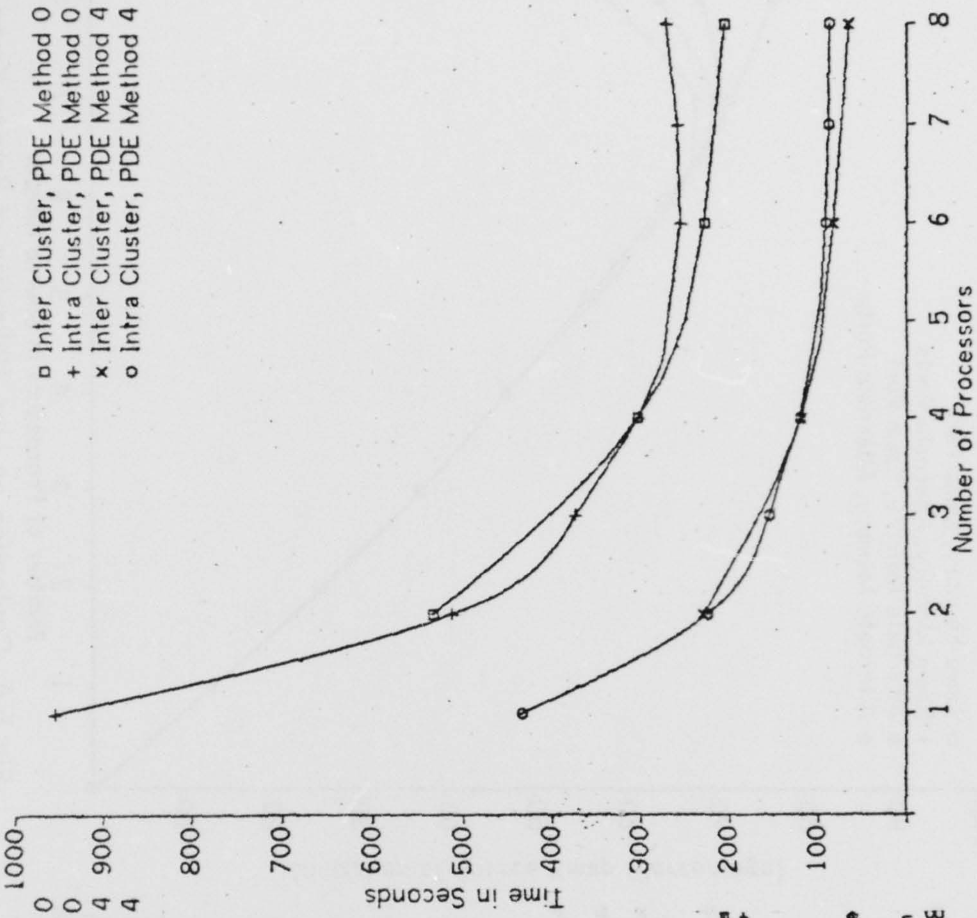


Fig 5.2 Intra & Inter Cluster Execution Time (All Mapped)

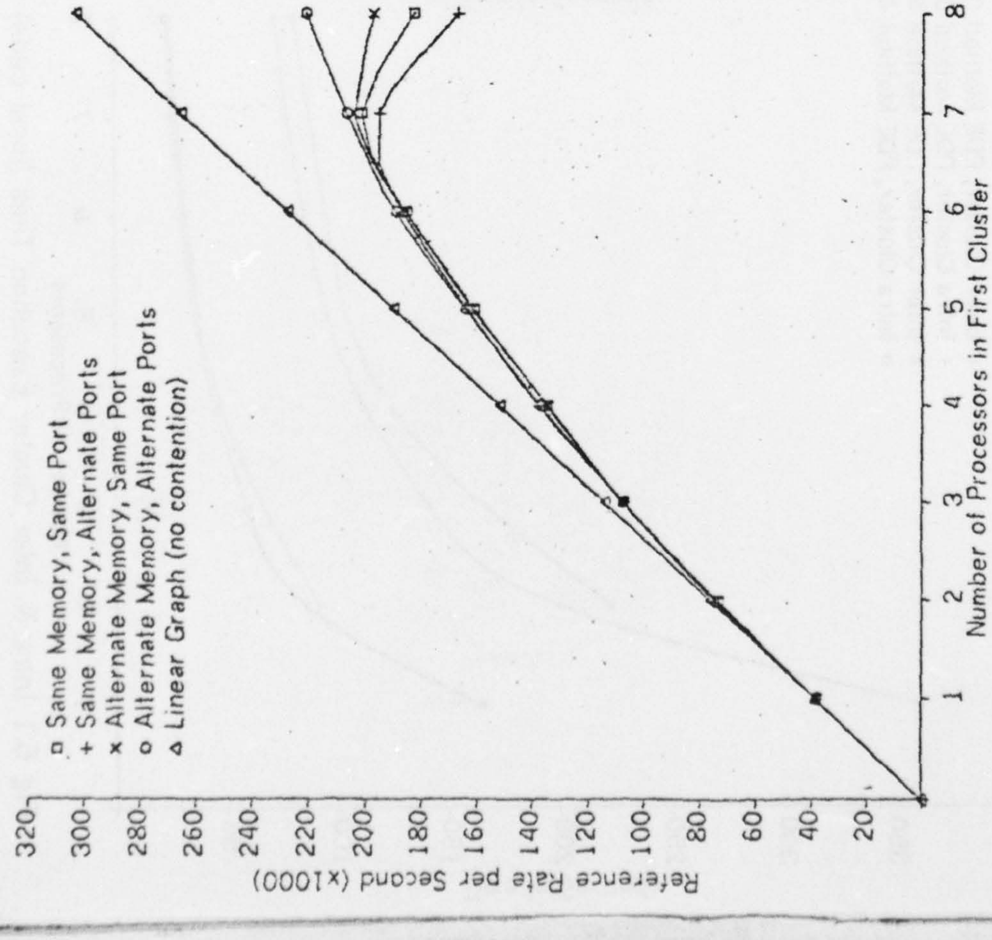


Fig 5.3 Contention on Linc, Reference Rate

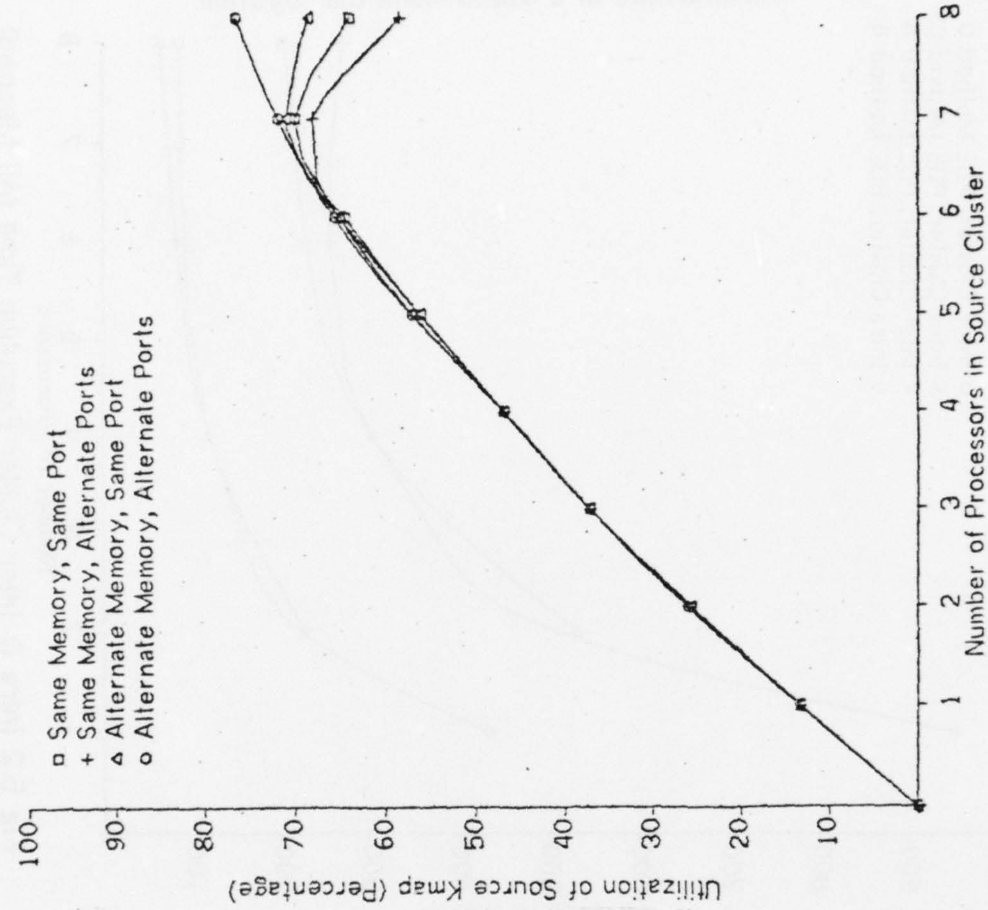


Fig 5.4 Contention on Linc, Utilization of Source Kmap

5. HARPY

P. Feiler

5.1. Introduction

HARPY is a speech recognition system developed at CMU. Its knowledge representation is a state transition network which is dynamically updated. The speech data is preprocessed (segmented) on the CMU-10B (CMUB). A search along a "few best" paths in the network is performed in parallel to recognize the speech. For more details on the Artificial Intelligence aspects of HARPY the reader is referred to [1].

5.2. HARPY on Cm*

5.2.1 Process Structure

The Cm* version of HARPY can recognize the DESK CALCULATOR (DESCAL) task. It has a 37 word vocabulary; the network consists of approx. 1000 states with an average branching factor of 4 (i.e., each node in the network has an average of 4 immediate neighbors). The program is decomposed into a set of cooperating processes. One *master* process communicates with the user's terminal, initializes the received input data, and co-ordinates a set of *slave* processes. The slave processes perform three functions according to the rules the master specifies - (network initialization, take one step in the network, prune the search tree and remember the result). For the *live* version of HARPY an *input* process is responsible for communication with the CMUB. The input process receives data from the CMUB and passes it to the master. The master process then prepares the data for the slaves.

5.2.2 Space Requirements

The space requirements for running the Cm* version of HARPY are as follows:

i. Each process requires 2K words of local code and 2K words for a local stack.

ii. The network's global data requires 26K words.

iii. The I/O process requires 2K words of buffers.

5.2.3 Implementation Alternatives

Several implementation alternatives are possible which affect the synchronization structure and hence the performance of the program. The private semaphores of the different processes can be implemented using a Kmap provided synchronization operation or by the method of busy waiting on a shared memory location. Exclusive access to the network can be guaranteed by using critical sections. The grain size of the lock used in critical sections affects the performance of the system. In the extreme cases, either the entire network is locked or an individual node is locked. Work on the network can either be distributed evenly in a predefined way, which does not require synchronization, or it can be dynamically partitioned into *workunits*. Workunits can be collected in a pool from which processes pick up the next available piece of work. In the latter case synchronization overhead is incurred. The choice of workunit size is influenced by the synchronization overhead and the variance and mean of the work demands of a workunit.

5.3. Experiments with HARPY on Cm*

Most of the experiments were performed on pre-recorded data. This data consisted of three phrases which correspond to approximately 6 seconds of speech. The fastest recognition of the speech achieved to date was 6.6 seconds or 1.2 times real time. In that case 6 slave processes were used; code, stack and private semaphores were local, and the unit of work were the best candidate nodes of the search through the network.

5.3.1 Speed-up with Multiple Processors

The following speed-up factors have been achieved on an 8 processor system (6 slave Pcs, 1 master Pc, and 1 utility Pc):

# of slaves	speed-up factor
1	1
2	1.87
3	2.64
4	3.17
5	3.44
6	3.60

An experiment on C.mmp - another multiprocessor at CMU - showed that the number of processors the HARPY algorithm can effectively use (i.e., can get more speed-up with) is bounded above by 7. Hence, increasing the number of processors beyond 8 or 10 will not improve HARPY's performance on Cm* unless the algorithm is changed.

5.3.2 Memory Distribution

An active HARPY slave process has the following memory reference distribution:

14%	Global data of the network
8-10%	Stack references
2%	Kmap operations
74-76%	Code references

An experiment was performed in which a comparison was made between the performance of a slave process executing remotely (i.e., fetching instructions from another Cm's memory) and the performance of a locally executing slave process. This experiment showed a factor of 2.5 difference in performance between the two cases, which corresponds to the variation in access time to the different levels of the memory hierarchy.

It is desirable to execute code in the local memory of a processor. The locality of the stack also has a strong influence on the performance of the program as can be seen in Fig. 4. The improvement in speed over accessing the stack remotely is about 12%. This figure can be even higher if the locality of the stack changes the memory reference pattern such that the Kmap becomes unsaturated.

5.3.3 Workload Distribution

The following HARPY experiment was performed. The experiment was divided into three parts; each time HARPY used two slave processes. The processes were arranged so that:

i. Both were executing remotely.

ii. One was executing locally and the other remotely.

iii. Both were executing locally.

The results of the experiment show the sensitivity of the algorithm to the speed variance of the different processors. It turned out that in the second case the performance lies between the values for the first and third cases. This is because part of the workload is distributed evenly, thus the slower processor was slowing down the faster processor at the synchronization points (the slave processes are synchronized 50 times per second of speech or 25 times for one real time second).

5.3.4 Synchronization and Cooperation

The Kmap supported operation *decrword*, which decrements a variable down to 0 and returns the old value of the variable, is a useful facility for implementing synchronization. For example for a set of processes to simultaneously push or simultaneously pop elements to or from a shared stack, the synchronization on the stack is embedded in the decrword operation on the stack pointer.

However using the Kmap to implement private semaphores is not recommended; A process busy waiting on a semaphore makes frequent references through the Kmap. Busy waiting on a local memory location is a better solution (see Fig. 2).

To guarantee exclusive access to a shared data structure, i.e., in order to update a node in the network, locks associated with the network must to be introduced. The unit of data to be locked can be the entire network, part of the network, or a node. A lock on the entire network causes slaves to queue up on the lock. The situation is improved by locking individual nodes (see Fig. 3). The latter scheme naturally increases HARPY's space overhead.

5.4. HARPY on CMU computers

The performance of HARPY with the desk calculator task on different machines at CMU is depicted in Fig. 1.

The C.mmp version of HARPY with one process, running on a PDP11/40, is slightly slower than the UNIX version on a PDP11/40. However with two processes it can run faster than real time. With more than seven processes no further speed-up can be obtained. This is inherent to the small network of the desk calculator task.

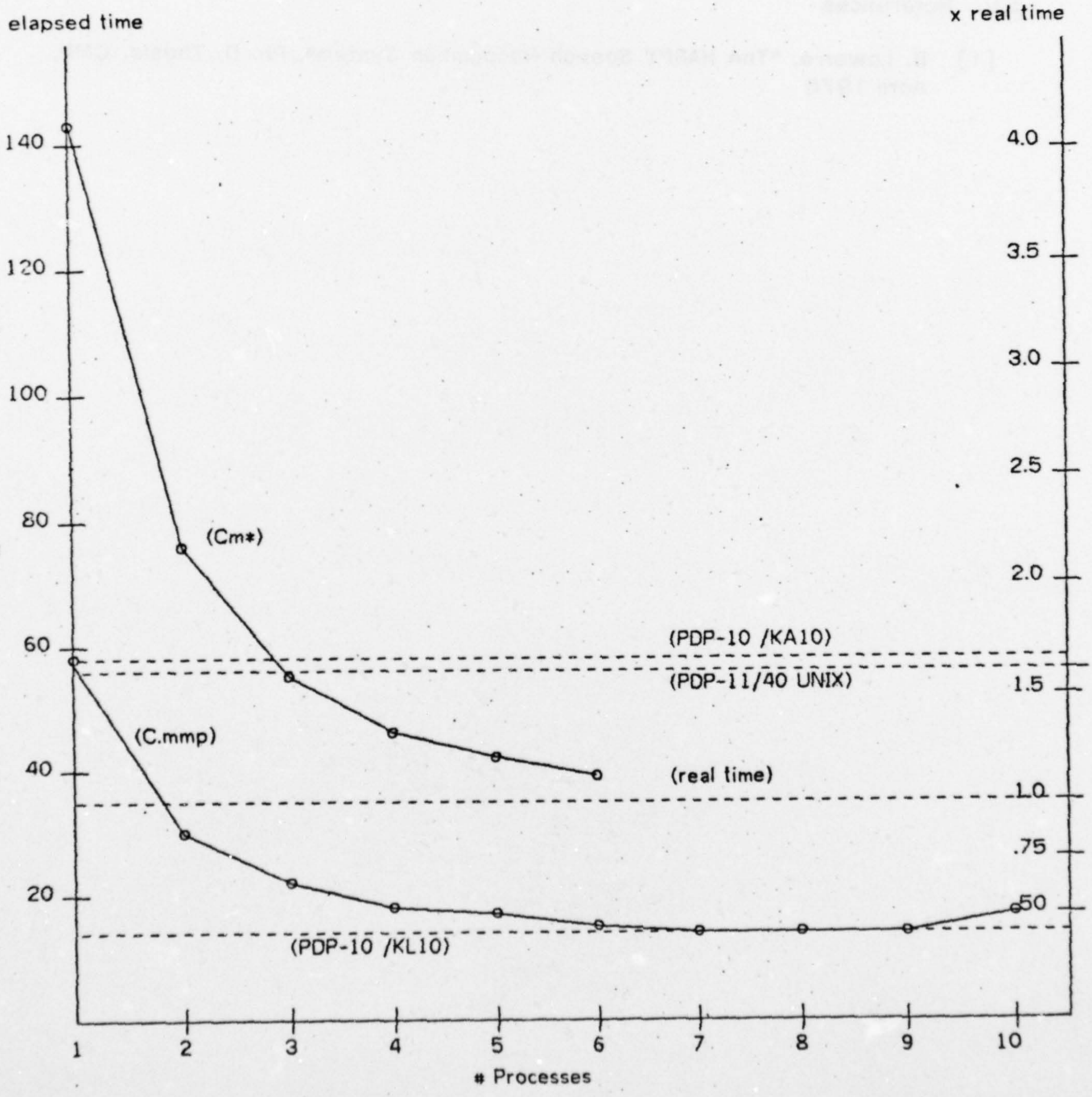
As expected, HARPY on Cm* is slower than HARPY on C.mmp. They differ in performance by a factor of 2.5, even though the basic processor speeds of the PDP11/40 and the LSI-11 differ by a factor of 3. The better performance of HARPY on C.mmp is mostly due to the more efficient implementation of synchronization mechanisms.

5.5. References

[1] B. Lowerre, "The HARPY Speech Recognition System", Ph. D. Thesis, CMU, April 1976

Fig.1 HARPY on CMU computers

DESCAL Task



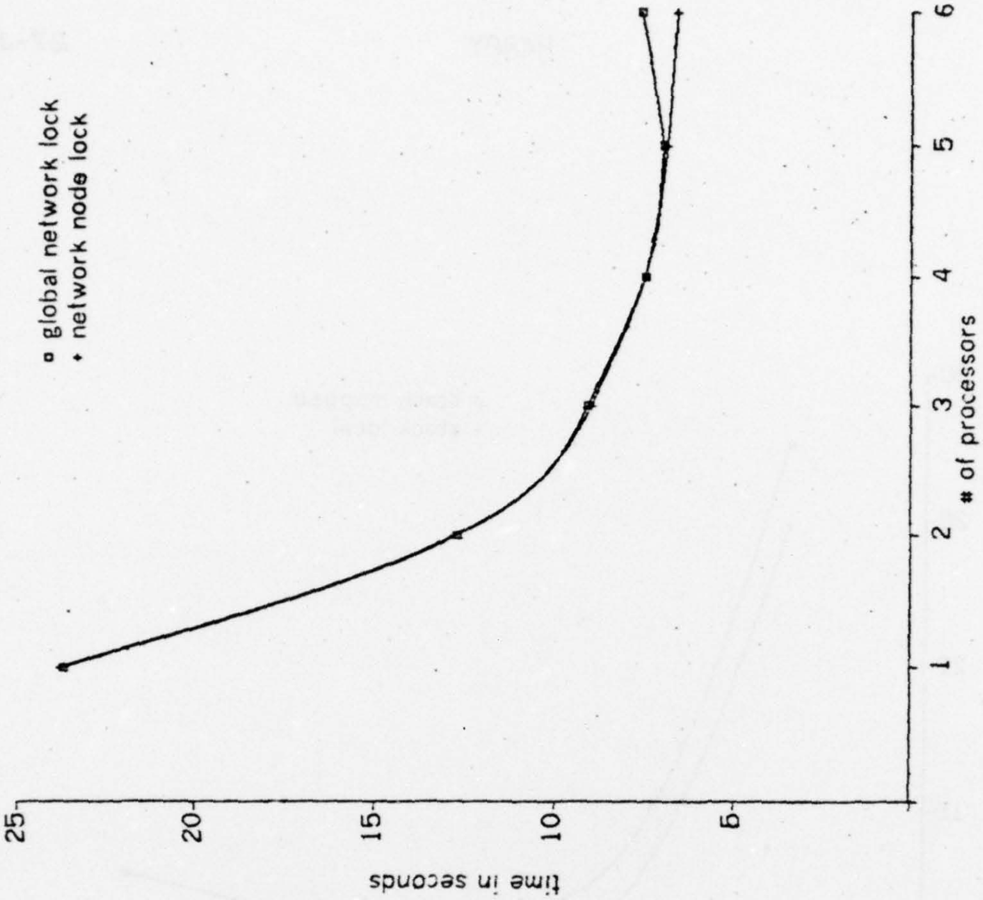


FIG.3 Network Locking

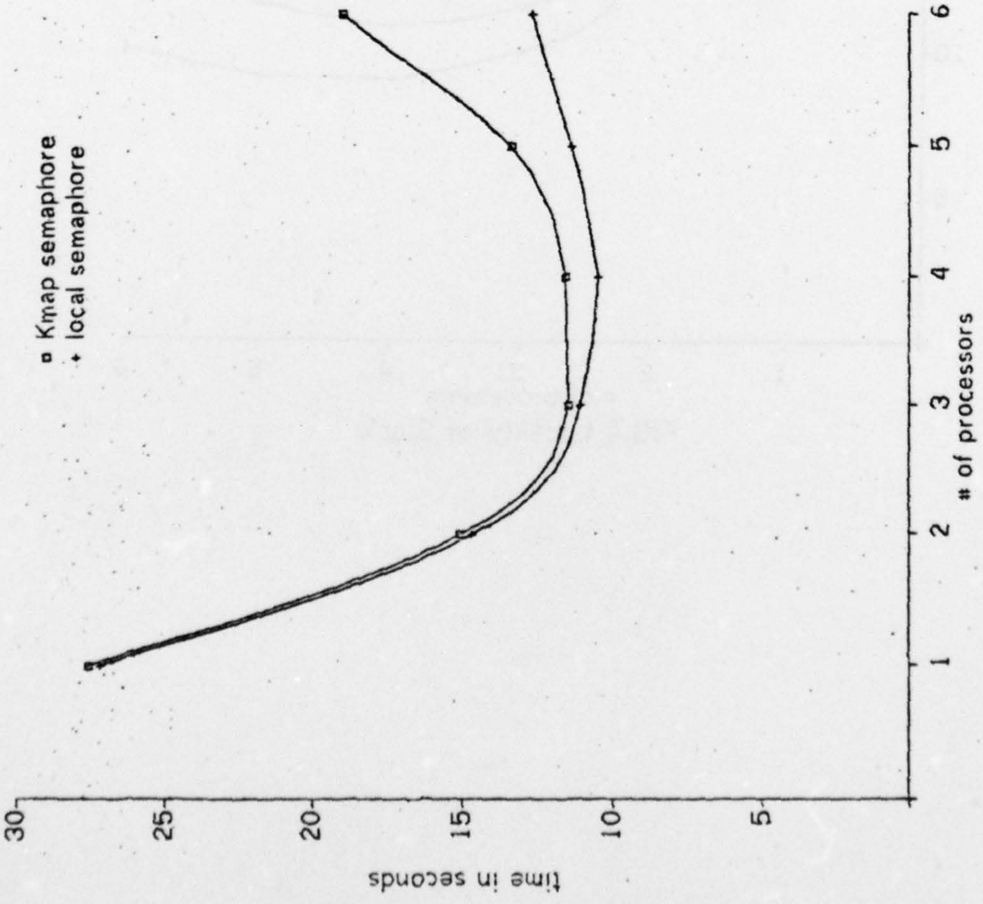


FIG.2 Private Semaphores

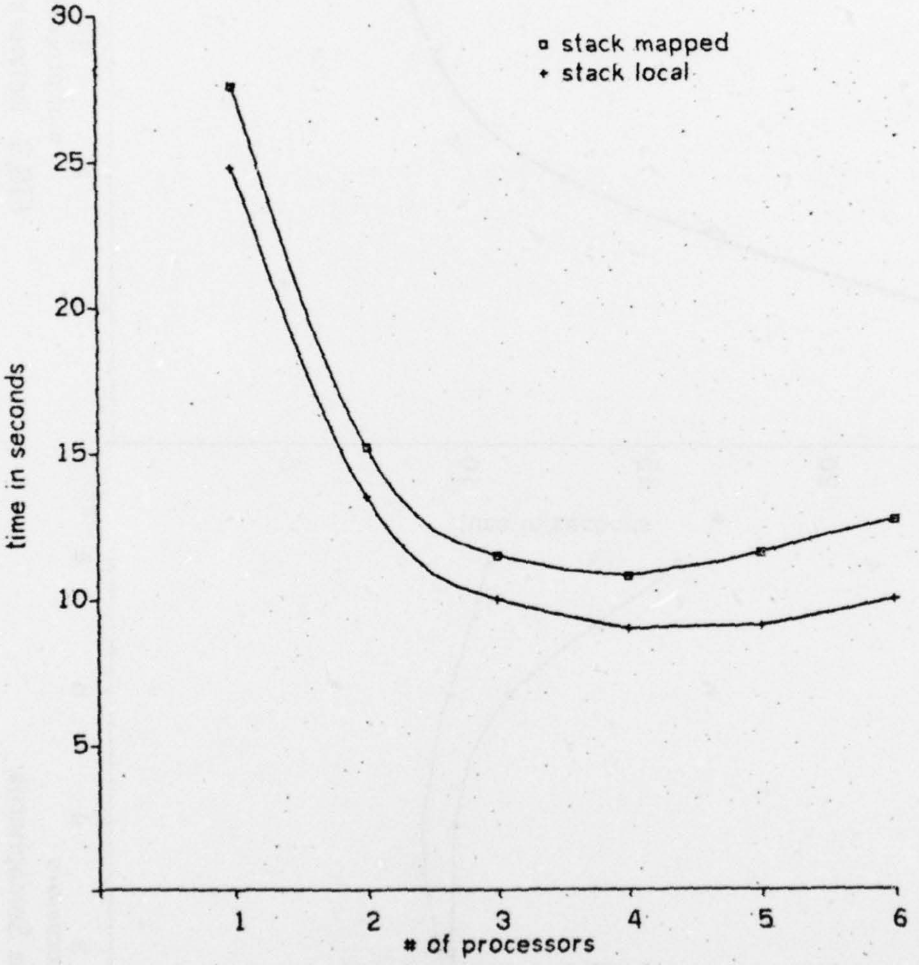


FIG.4 Locality of Stack

6. Algol 68 on Cm*

6.1. Investigators

Peter Hibbard, Andy Hisgen, Tom Rodeheffer (with assistance gratefully received from Paul Knueven and Bruce Leverett).

6.2. Introduction

The programming language Algol 68 is being implemented on Cm* for the following reasons:

- To provide a language for general purpose programming applications and for software experiments;
- To study the performance of a technique for decomposing programs automatically to execute concurrently on several processors, and to study the suitability of the Cm* architecture for this technique.

6.3. Algol 68

The language we are implementing is a semantically rich subset of Algol 68. The subset has official recognition by the International Federation for Information Processing. It has been derived from full Algol 68 by the omission of infrequently used facilities, and by restrictions to simplify compilation; however, it remains somewhat more powerful than PL/I. In order to take advantage of the parallel architectures on which it runs, the official subset has been extended by including several methods of specifying concurrent execution and synchronisation of subtasks [1].

6.4. C.mmp Implementation

An implementation of subset Algol 68 has been running on C.mmp for about 18 months. It comprises:

- A single-pass compiler. This is written in a common subset of Bliss/10 and Bliss/11, and thus may be used as a cross-compiler on the PDP10. It runs on a single processor, and in the PDP11 version occupies about 20K words of code.

- A run-time support system. This provides the standard functions, I/O routines and the primitives for scheduling and synchronising concurrent subtasks.

- A linker (still being written). It will also act as a cross-linker from the PDP10 to the PDP11.

Extensive measurements of the performance of the run-time system have been made [2].

6.5. Cm* Implementation

The project of bringing up the Cm* version has been split into five phases:

- The installation of the C.mmp run-time system on Cm*, operating on top of a small kernel, and driven by the PDP10 cross-compiler;

- The evaluation of the performance of the system, and its comparison to the C.mmp version;

- The installation of a number of modifications, mainly into the run-time system, to support the automatic decomposition of sequential tasks into concurrent subtasks;

- The evaluation of the performance and effectiveness of these modifications, with a view to incorporating some of them into Kmap microcode;

- The installation of the compiler for the resulting system on Cm*.

The first two of these tasks have been completed, and the second two are now in progress. At some stage in the evaluation a compiler will be installed for general programming use.

In the following sections we describe the functions of the kernel, give the measurements which have been made on the basic run-time system and describe the modifications which are being made.

6.6. The Kernel

In order to avoid interference from operating system overheads while collecting performance statistics, the run-time system runs upon a small, special-purpose kernel, which provides basic support for interrupt and I/O handling, segment allocation and swapping, bootstrapping and the collecting of performance statistics. Only very minor modifications to the run-time system executing on C.mmp have been required to get it to execute above this kernel, and similar minor modifications will be required to get it to execute above the Cm* Operating System.

6.7. Performance of the Basic Run-time System on Cm*

Measurements of the locality of memory references for several Algol 68 programs show that 80-85% of the references are to local memory, a reasonably high figure considering that no efforts need be taken by the programmer to achieve this, and the compiler has no knowledge of the mapping properties of the Cm* architecture. Measurements have also been made on several programs which make use of the parallel processing facilities of Algol 68. The speed-up obtained for polyevent, a program which manipulates branching data structures, executing on several processors is given in Figure 2 of the Introduction.

6.8. Automatic Decomposition of Sequential Programs

As in other languages, the parallel processing facilities in Algol 68 require the explicit decomposition of a program into relatively large-grain subtasks, and their explicit synchonisation using semaphores. Whilst some advances have been made in simplifying this decomposition, it still remains the case that much potential small-grain parallelism cannot be exploited due to the lack of language features to support it. On Cm*, with relatively low-power processing units, the importance of obtaining large degrees of parallelism is high.

The modifications which are being studied to provide for automatic decomposition into small-grain subtasks comprise a software implementation of multiple parallel instruction pipelines, in which the instructions are the primitive actions of the Algol 68 run-time system (e.g. floating point operations, array indexing and other vector operations and assignments of large values). These actions are executed by "slave" processors on behalf of the "master" processor which is placing the actions in the pipeline. The overall control of the pipeline is distributed throughout all the processors, and processors may alter their designation according to the load.

Investigations of this system which are in progress include:

- Measurements of the performance of the system as a function of several parameters, such as the number of actions in the pipeline, the degree of interdependency of the actions and the critical section overheads;
- Assessment of the improvements to be expected by encoding certain of the actions which maintain the pipeline in Kmap microcode;
- Design of appropriate data-structures and the run-time routines which manipulate them;
- The measurement of the performance of the system when used with practical algorithms.

Several of the results which have been obtained are tabulated below.

Tables

The tables show the speed-up obtained by operation pipelining in the Algol 68 run-time system, and the percentage of the time that the queue of operations in the pipeline is locked for synchronisation. In order to determine the effects of the pipelining overheads on processor utilisation, the relative cost of pipeline operations to floating point operations was varied. This was done by keeping the pipeline costs fixed and slowing the floating point operations by factors of 1 (no delay), 2, 4 and 10. The high percentage of the time that the queue is locked shows that slave dispatching then becomes the principle overhead. Some distortion of the time the queue was locked for small numbers of slaves occurs due to the perturbations caused by the sampler which was collecting the statistics; the perturbations have insignificant effects on the execution times. These preliminary results indicate that a five-fold speed-up might be expected if the pipelining overheads can be reduced by rewriting them in Kmap microcode.

Speedup of Sequential Romberg Integration

number of processors	1	2	3	4
effective processors	1.00	1.81	1.99	1.86
% time queue locked	11	27	48	49

Speedup of Fast Fourier Transform (inner loops)

Standard floating-point operation cost

number of processors	1	2	3	4
effective processors	1.00	1.84	1.78	1.72
% time queue locked	10	24	12	24

Twice cost for floating-point operations

number of processors	1	2	3	4
effective processors	1.00	1.88	2.52	2.35
% time queue locked	26	23	14	50

Four times cost for floating-point operations

number of processors	1	2	3	4	5	6	7
effective processors	1.00	1.92	2.60	3.27	3.35	3.20	3.24
% time queue locked	26	27	38	37	55	52	65

Ten times cost for floating point operations

number of processors	1	2	3	4	5	6	7
effective processors	1.00	1.95	2.90	3.65	4.26	4.48	4.35
% time queue locked	27	30	16	32	34	47	53

References

[1] Peter Hibbard, Parallel Processing Facilities, New Directions in Algorithmic Languages, IRIA, 1976.

[2] Peter Hibbard, Paul Knueven, Bruce Leverett, A Stackless Run-time Implementation Scheme, Fourth International Conference on the Design and Implementation of Algorithmic Languages, Courant Institute, 1977.

7. Dynamic Reconfiguration

I. Durham

7.1. Current Status

Dynamic Reconfiguration is divided into two parts: Dynamic Recovery of system elements and Dynamic Removal of system elements. Subject to the restrictions described in the outline below, Dynamic Recovery is complete and operational. Dynamic Removal of system elements is partially designed. Dynamic Recovery is routinely executed as part of system initialisation. Loading the operating system consists of bringing up a single Cm and dynamically reconfiguring in the remainder of the Cms in the Cluster.

7.2. Dynamic Recovery Facilities

Dynamic Recovery provides for recovering an entire cluster, individual Cms, individual Mps of Cms, and individual Pcs of Cms. Recovery of a system element means that the resource provided by the element becomes available.for use by the system.

Cluster recovery involves a single cluster. The first module in any cluster must be specially initialised before it can execute Cluster Recovery. Cluster Recovery involves attempting to recover each additional Cm in the cluster. It is possible to specify a subset of Cms to be ignored during cluster recovery.

Individual Cm recovery first checks that the Slocal for the Cm responds to references. If it does respond, then both the Pc and Slocal are initialised (brought into a known, dormant state). The Cm's Mp is recovered first and then the Pc is recovered.

Recovering Mp involves creating the Local Mp Manager's data structures, sizing memory, and initialising the Cm's local BitString (which is used to record allocation of blocks of physical memory). The Local Mp Manager's structures are then linked in as one more element in the distributed Cluster Mp Management data structure.

To recover a Pc means to set up all of the Cm's private structures that are used by the Operating System and uniquely associate them with this Pc. The PCL

(Primary Capability List) is created first and capabilities for existing operating system data structures are copied into it. Then other segments used by the Cm are created; these include the Control Stack, Control Stack interrupt vectors, and Slocal segment.

Dynamic Recovery has been used to recover a Cm into an already running cluster.

7.3. Overview of Initialisation

An over-simplification of system initialisation would be to say that all Cms in all clusters are recovered. More specifically, the strategy is that the first Cm in each cluster is individually initialised. Once initialised, this first Cm executes Cluster Recovery to bring up the other Cms in the cluster. The very first Cm in the world is special in that during its initialisation certain system-wide data structures are created. In particular, the Segment Directory is created and initialised.

Initialising (Recovering) clusters other than the first involves three stages:

(1) Locating a Cm in the remote cluster which is capable of performing initialisation,

(2) Depositing 'first Cm' initialisation code into it and starting it, and

(3) Recovering all other Cms in the remote cluster. This last stage is identical to that performed in the very first cluster, since Cluster Recovery operates only on the 'local' cluster.

8. Cm* Host System

D. Scelza

8.1. Introduction

This report gives a brief description of the Cm* Host System. A brief outline of both the software and hardware is provided. The Cm* Host System provides all communication with the Cm* machine at this time. In this role the Host System is a most valuable tool.

8.2. Overview

The Cm* Host is a line oriented system. Its primary function is to facilitate intercommunication between a large number of serial lines. Lines, both those that connect to terminals and those that connect to other computers, are treated in a uniform manner.

The uniform treatment of these lines means that a terminal can look just like a Computer Module and, more importantly, a Computer Module can look just like a terminal.

8.3. Use

The primary use of the Cm* Host System is for the control of Computer Modules. A user can log into the Host and assign resources to be used. These resources generally include one or more Computer Modules. After a user has assigned resources he can have the Host execute commands that let him communicate with these resources. He can be put into a direct cross-patch mode with his assigned resource. This gives him the ability to talk to his Computer Module just as if his terminal were directly connected to that module. The user can also load programs into his Computer Modules. The loading can be done from either the DECtape or from the PDP-10. As long as a user is logged in and has resources assigned no one can interfere with his use of those resources. When a user logs off of the Host all of the resources that he had assigned are then free to be used by other users.

8.4. Hardware

The Cm* Host System runs on a Digital Equipment Corporation PDP-11/10. The PDP-11/10 is equipped with 28K of core memory and numerous peripherals devices. These devices include:

a) A DECtape controller with four DECtape drives.

b) A DJ11 16 line serial line multiplexer.

c) 2 DR11-C parallel interfaces.

d) Three local terminals.

e) A host link to the C-MU Front End. (This is a multiplexed line along which there can be any number of virtual terminals.)

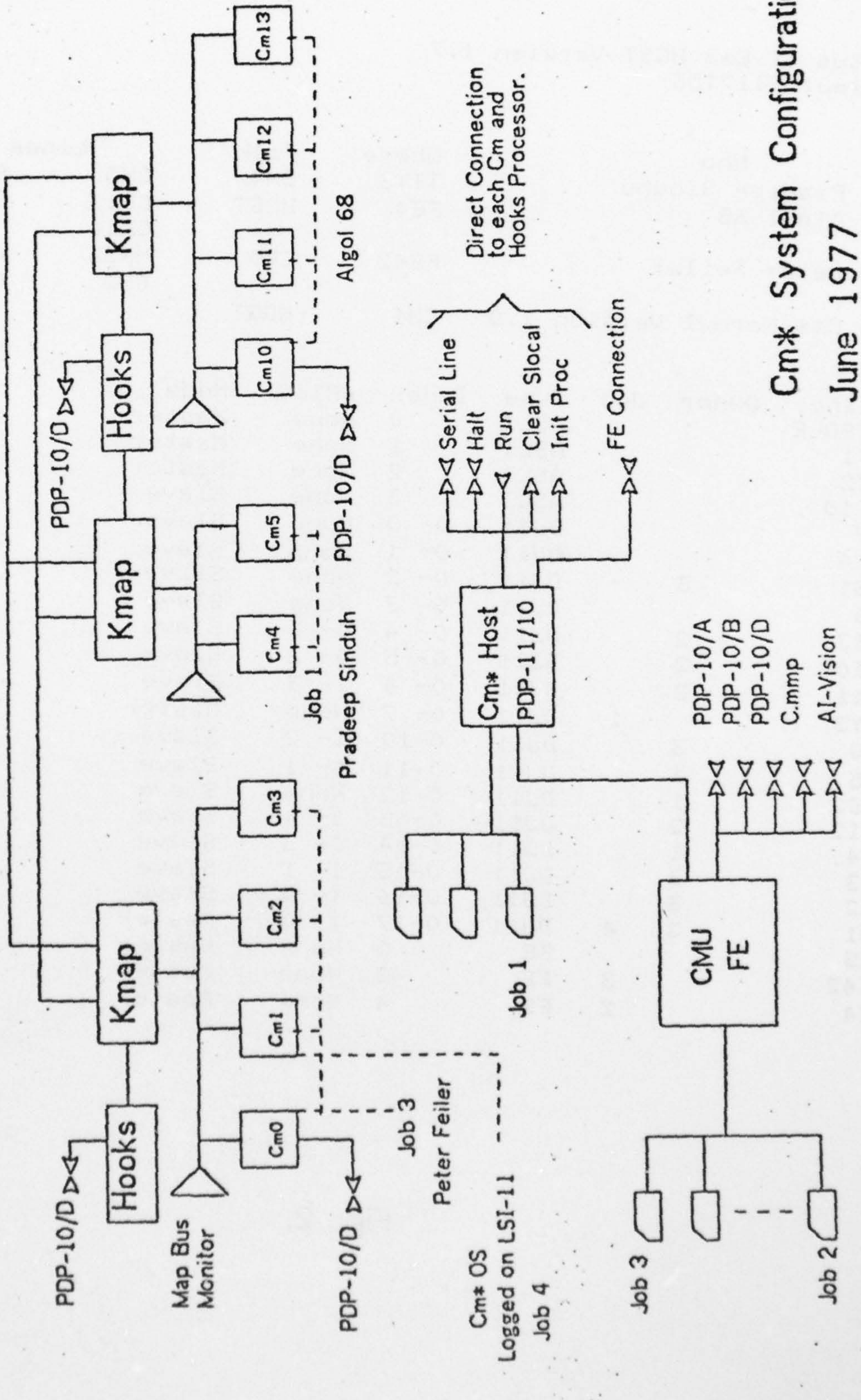
f) A terminal link to the C-MU Front End.

8.5. Software

The Cm* Host System consists of about 16K of BLISS-11 code which runs on the PDP-11/10 along with a small amount of support software which runs on the PDP-10's.

8.6. Example

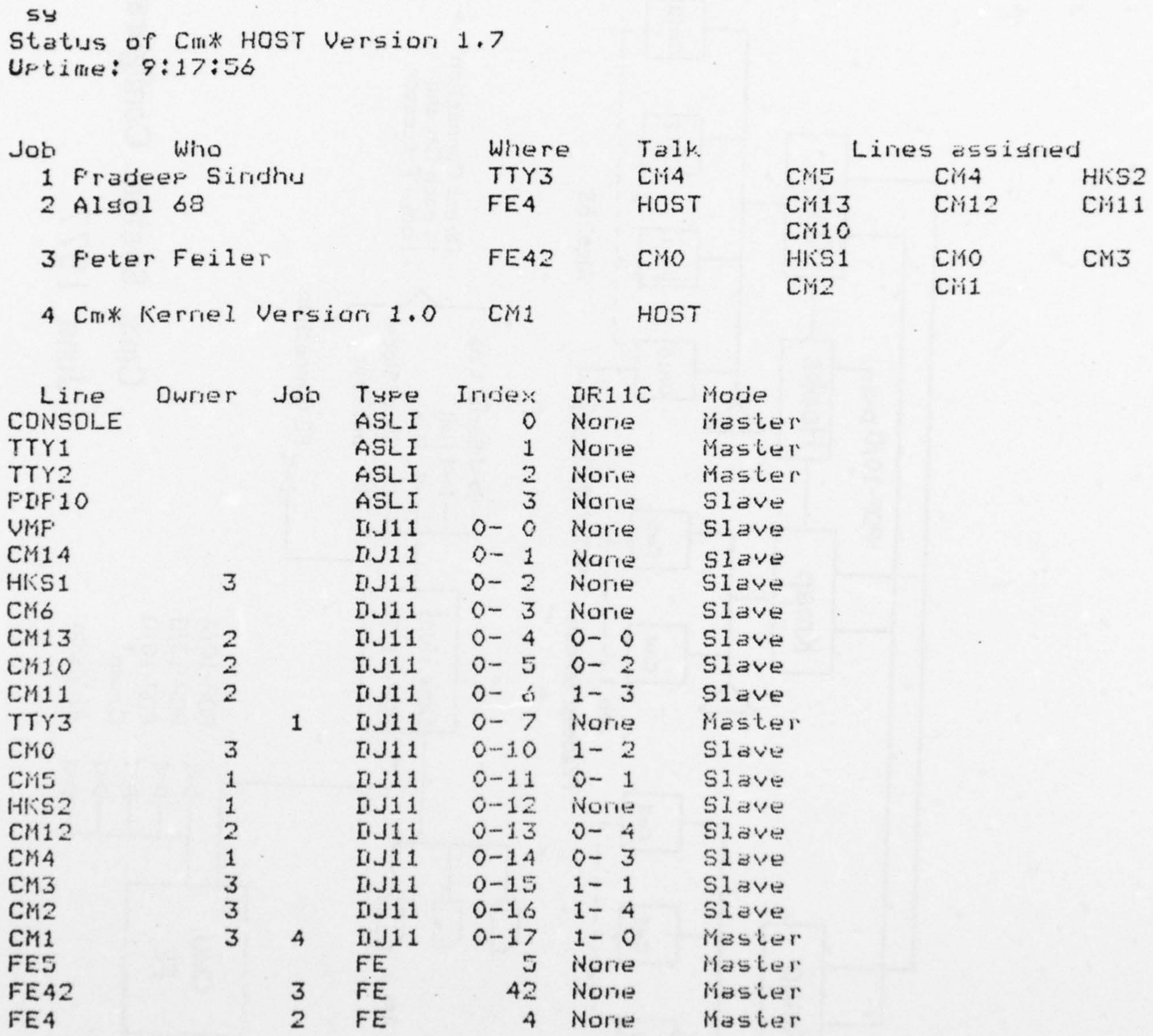
The two figures below give an example of the use of the Host. The diagram of the Cm* system (Fig. 1) has been annotated to correspond to the system status report (Fig. 2). We see that there are four jobs logged into the Host. Three of these jobs are on terminals (jobs 1,2, and 3), while the fourth job is logged in from a Computer Module. The diagrams also show that the jobs 1,2, and 3 have each been allocated a set of resources that only they can access.



Cm* System Configuration

June 1977

Fig. 1.



```
 sy
Status of Cm* HOST Version 1.7
Uptime: 9:17:56

Job         Who                      Where      Talk          Lines assigned
  1 Pradeep Sindhu                  TTY3       CM4        CM5        CM4        HKS2
  2 Algol 68                        FE4        HOST       CM13       CM12       CM11
                                                          CM10
  3 Peter Feiler                    FE42       CM0        HKS1       CM0        CM3
                                                          CM2        CM1

  4 Cm* Kernel Version 1.0          CM1        HOST


  Line    Owner   Job   Type   Index   DR11C   Mode
CONSOLE                 ASLI       0   None    Master
TTY1                    ASLI       1   None    Master
TTY2                    ASLI       2   None    Master
PDP10                   ASLI       3   None    Slave
VMP                     DJ11    0- 0   None    Slave
CM14                    DJ11    0- 1   None    Slave
HKS1         3          DJ11    0- 2   None    Slave
CM6                     DJ11    0- 3   None    Slave
CM13         2          DJ11    0- 4   0- 0    Slave
CM10         2          DJ11    0- 5   0- 2    Slave
CM11         2          DJ11    0- 6   1- 3    Slave
TTY3                1   DJ11    0- 7   None    Master
CM0          3          DJ11    0-10   1- 2    Slave
CM5          1          DJ11    0-11   0- 1    Slave
HKS2         1          DJ11    0-12   None    Slave
CM12         2          DJ11    0-13   0- 4    Slave
CM4          1          DJ11    0-14   0- 3    Slave
CM3          3          DJ11    0-15   1- 1    Slave
CM2          3          DJ11    0-16   1- 4    Slave
CM1          3      4   DJ11    0-17   1- 0    Master
FE5                     FE         5   None    Master
FE42                3   FE        42   None    Master
FE4                 2   FE         4   None    Master
```

Fig. 2.

9. Reliability and the Auto-diagnostic

H. Bellis

Experimental systems typically have poor reliability and it was initially feared that Cm*, with its many varied components, might be especially unstable during its period of development. In an attempt to monitor the frequencies of transient errors and to detect hard errors as soon as possible, an automatic diagnostic system (the *Auto-diagnostic*), has been developed.

The Auto-diagnostic program resides in one of the modules of the Cm* system. After requesting the time and date from an operator, the Auto-diagnostic logs into the Cm* Host and assigns any Cms which are currently free (i.e., not assigned to any other user logged into the Host). (See Section 8.) The Auto-diagnostic will return Cms to the Host when so requested by a user.

The Auto-diagnostic loads a standard diagnostic program, from the Host's DECtapes, into each module. When the diagnostic has completed a run in a module, the Auto-diagnostic replaces it with the next diagnostic in the testing sequence. This sequence currently consists of four diagnostics:

i. A memory diagnostic. (DEC's standard DZKMA for LSI-11's).

ii. An instruction set diagnostic. (DEC's standard DVKAA for LSI-11's).

iii. An interrupt and trap diagnostic. (DEC's standard DVKAD for LSI-11's).

iv. An Slocal diagnostic.

The Auto-diagnostic monitors the output of the Cms it is testing and produces a statistical error report for each Cm. The engineer can examine this report each morning to determine which modules are in need of attention.

The Auto-diagnostic is run whenever there are free Cms which may be tested. (In the future, the Host may be updated to automatically load the Auto-diagnostic whenever possible.)

The statistics gathered during that past few months (i.e., since the Auto-diagnostic has been operational) are summarized below.[2]

Transient Errors

7-May-77 to 2-Jul-77

	Memory	Inst.Set	Traps	Slocal	Total	HH:MM	MTBF
Cm0	14	0	1	0	15	162:41	10:51
Cm1	2	0	0	1	3	188:18	62:46
Cm2	1	4	4	4	13	209:45	16:08
Cm3	0	0	0	4	4	208:27	52:07
Cm4	0	0	0	1	1	157:55	157:05
Cm5	0	4	0	1	5	149:52	29:58
Cm10	0	2	0	1	3	160:12	53:24
Cm11	8	0	0	0	8	207:13	25:54
Cm12	4	0	1	0	5	218:23	43:41
Cm13	1	0	3	0	4	102:15	25:31
Total:	30	10	9	12	61	1765:01	28:56

[2] These statistics are preliminary and should not be taken as an indication of the reliability of a fully stabalized system. Further data collection and analysis are required.